A Versatile Tuple-Based Optimization Framework

Rietveld, K.F.D.

A Versatile Tuple-Based Optimization Framework

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 10 april 2014
klokke 16:15 uur

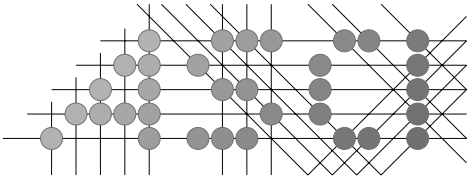
door

KRISTIAN FRANS DANIËL RIETVELD

geboren te Vlaardingen
in 1984

Samenstelling promotiecommissie:

Promotor:	Prof.dr. H.A.G. Wijshoff
Overige leden:	Prof.dr. B.H.H. Juurlink (Technische Universität Berlin) Prof.dr. F. Arbab (CWI, Universiteit Leiden) Prof.dr. F.J. Peters Prof.dr.ir. E.F.A. Deprettere Prof.dr. J.N. Kok



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 301.

A Versatile Tuple-Based Optimization Framework
Kristian Frans Daniël Rietveld
PhD Thesis, Universiteit Leiden
ISBN: 978-90-8891-855-1
Printed by: Proefschriftmaken.nl
Published by: Uitgeverij BOXPress, 's-Hertogenbosch

Aan Ginny

Voor mijn ouders

What I cannot create, I do not understand.

- Richard P. Feynman

Contents

CHAPTER 1

Introduction

In imperative programming, the computation to be carried out is specified step by step. Declarative programming languages, on the other hand, allow the specification of *what* data must be retrieved, but not how. Naturally, these two different programming paradigms give rise to different manners by which codes expressed in these paradigms are translated to code that can be executed on a Central Processing Unit (CPU) and different manners by which these codes are optimized. Whereas code optimization for imperative programming languages focuses on reordering the steps by which the computations are carried out without affecting the final results of these computations and selecting efficient instructions to encode these computations for a particular CPU code, optimization for declarative programming languages focuses on determining an efficient execution plan for retrieving the specified data.

Application programs are in general written in an imperative programming language. Declarative programming languages are used to write codes, or queries, that specify data to be retrieved from a Database Management System (DBMS). Such queries can be performed by application programs and the retrieved data can be further processed by these application programs. Applications that perform such requests are referred to as *database applications*. In database applications, the data processing code is written in an imperative programming language and the data retrieval request is written in a declarative programming language. As a consequence, data processing code and data retrieval code undergo different and independent optimization procedures. For example, queries are optimized and executed in a DBMS, independent from the application code that further processes the result data.

In this thesis, a novel approach is presented for the optimization of data-intensive applications. This approach is implemented by the *forelem* framework and solves three important problems of existing approaches. Firstly, this framework unifies the programming of transactional (database) applications and the programming of other kinds of applications, such as high-performance parallel computational codes. An important difference between these two forms of programming is the kind of optimization that is performed in a DBMS and that is per-

formed on application codes by traditional optimizing compilers. The DBMS (query) optimizations are necessary to efficiently retrieve the desired data from a database, especially when the data set does not fit in main memory, while the traditional optimizations performed on application codes are vital for the generation of efficient machine code. The *forelem* framework unifies these seemingly distinct fields of programming by expressing queries as a series of array accesses governed by simple loop control, which are subsequently optimized by traditional optimizing compiler techniques accomplishing results similar to query optimization.

Secondly, the *forelem* framework provides a solution for the (semi-)automatic problem-specific optimization process for applications, which runtime is very much dependent on the underlying characteristics of the problem to be solved. Problem-specific optimizations often consist of the selection of a good data layout or data storage method. In general, compilers do not have the capability to optimize data layout or storage. Although techniques to extend compilers with the ability to optimize data structures have been researched, they have not yet found a widespread use. With the *forelem* framework, a universal approach is introduced for the optimization of an application's data layout and storage. By incorporating details about the data access performed by the application into the optimization process, the application and its data access method can be synchronized. This synchronization leads to a better alignment of the application's computational loops with the order in which data is accessed.

Thirdly, the unification of transactional programming and other kinds of programming enables the vertical integration of application code and data access frameworks. Applications typically access data through a framework that abstracts away peculiarities of accessing a particular file format, database system or distributed file system. Such frameworks inhibit optimizing compilers from potentially optimizing data access as performed by an application. The *forelem* intermediate representation provides a generic way for expressing data access, based on series of array accesses and simple loop control. In vertical integration, the data access operations that are performed through a data access framework, are expressed in this generic intermediate representation. As a result, the data access code is combined with the surrounding application code in the optimization process. Traditional analysis methods, such as Def-Use analysis, will detect and eliminate data access of which the results are unused, or will detect related data accesses that can be combined. For database applications, methods that optimize both the application and data access codes (in the form of queries) have been proposed [66, 37, 21]. However, because both codes are kept separate, these methods often rely on pattern recognition to detect possible code segments where a specific optimization can be applied. In the *forelem* framework generic optimizations can be applied, which unlock many more potential optimization opportunities.

Since the *forelem* framework was initially envisioned for database applications, it considers data to be stored as (multi)sets of tuples. So, the *forelem* framework operates on a tuple space. Due to the generic nature of the *forelem* intermediate representation, the framework is applicable to many different application areas. Examples of these application areas are: (1) vertical integration of database applications: queries in a database application are replaced with code segments that

evaluate these queries and directly access a data store, subsequently, the application and data access codes are optimized together (Chapter 7); (2) reduction of energy consumption: a reduction in energy consumption up to 90% can be obtained by the application of vertical integration coupled with aggressive optimization (Chapter 2); (3) exploration of the optimization search space: the *forelem* intermediate representation expresses data access using simple loop control, which enables re-use of traditional compiler loop transformations and in conjunction with this new transformations and heuristics are to be developed, as well as methodologies to effectively explore the optimization search space (Chapter 10); (4) data reformatting: an optimization process incorporating details about the application code and associated data access is able to optimize storage layout and format for this particular application (Chapter 9); (5) Big Data: new optimization techniques for Big Data applications can be devised using distributed *forelem* loops combined with automatic optimization of the data distribution and layout (Chapter 12); (6) universality: data access expressed in many different methods, such as SQL or MapReduce, can be translated to the *forelem* intermediate representation and vice versa.

This thesis describes a versatile tuple-based optimization framework. A versatile framework, because it is capable of optimizing traditional imperative codes (such as sparse matrix computations) as well as declarative codes (such as database queries). Although the framework is tuple-based due to initially being designed for the optimization of database applications, tuples are especially suited as an elementary data representation because they are the most fundamental objects that allow multiple values that are related with each other to be coupled. All data structures can be represented as tuples, in fact, computer memory can be expressed as tuples by creating pairs of address and an associated value. From a representation of the data in the form of tuples, many different data layouts can be automatically generated.

The first part of this thesis discusses the application of the *forelem* framework to database applications. The unification of transactional and imperative programming that is achieved by this framework enables the vertical integration of application code and data access frameworks. In past research also methods have been described that tried to overcome the division between DBMS (declarative) and application (imperative) codes. Such methods included object-oriented data systems integrated with object-oriented programming languages and specific database programming languages. The importance of this unification has been raised in the literature. For example it has been asserted that compilers for database programming languages must be extended to include database-style optimization in order to produce high performance codes [62] and a call has been made for a single expressive intermediate language instead of the use of specific representations for database queries and generic program codes [38]. Unfortunately, such approaches failed to get traction in the database community and it is still common practice to specifically program the database access code, that places requests for data retrieval, while keeping the division in place [37]. A major advantage of the approach described in this thesis is that it is not necessary to rewrite existing database application code in order to benefit from vertical integration.

As an initial target of our vertical integration efforts, we selected an important class of database applications: web applications. Web applications are these days ubiquitous and empower the modern, interactive World Wide Web. The increase of such web applications led vendors to significantly scale up their banks of web and database servers in order to handle all incoming requests. This has resulted in ever increasing complexity of distributed server architectures. Optimization of such systems not only leads to faster response times, but also to the ability to handle similar loads with a smaller amount of servers. For the implementation of this vertical integration, the *forelem* framework is introduced. The *forelem* framework is ideally suited to support an integrated, holistic optimization process as will be discussed in the first part of this thesis.

As will be described in Chapter 2 of this thesis, this global integrated optimization process is capable of aggressively optimizing web applications, eliminating up to 90% of the instructions that need to be executed without affecting the final result, resulting in a tremendous increase in performance. The further chapters in Part I of this thesis describe a framework within which this optimization process can be carried out. The main constituent of this framework is the *forelem* loop, which specifies iteration of a subset of a multiset of tuples, and transformations that can be carried out on these loops, as described in Chapters 3, 4 and 5.

The generic nature of the *forelem* framework also allow the developed techniques to be applied in other application domains. This is the focus of Part II of this thesis. In this part, the use of the *forelem* loop is explored for the optimization of irregular applications. To accomplish this, the computation to be optimized is expressed in terms of tuples: the data that is operated on is translated to tuples, computations are translated to *forelem* loops processing these tuples. Chapter 10 focuses on sparse matrix computations. By expressing these in terms of tuples, the computation is reordered as well as the way in which tuples are stored is reorganized. As will be described in the chapter, this results in a large search space of different variants of the same computation. In Chapters 11 and 12, the techniques are further generalized to irregular applications and extended to be capable of expressing distributed computations.

In this chapter, the necessary background knowledge for this thesis is introduced, as well as work that is related to this thesis. This background knowledge has been divided into three fields, Optimizing Compilers, Database Systems and Sparse Computations, that will be discussed in turn.

1.1 Optimizing Compilers

Code written in traditional compiled languages such as C and Fortran, is directly translated to machine code for a particular target architecture. The performance of this resulting machine code can be optimized by making use of optimizing compilers, that perform code transformations that are expected to improve the performance. In general, two levels can be distinguished at which optimization may take place. The optimizations can be performed on the structure of the original code, in which loops are still explicitly exposed. Examples of such transformations are constant propagation, common subexpression elimination and loop

transformations such as loop blocking and loop interchange. The other level at which optimization can be performed is the code generation level, where executable code for a particular target is generated from an internal representation of the program code. At this level, optimizations are performed such as instruction selection and scheduling. These optimizations are generally very specific to the instruction set architecture (ISA) of the target architecture.

Within the *forelem* framework, optimizations are defined that operate on the loop structure. These optimizations are based on traditional optimizing compiler transformations that are described in the literature. These transformations need slight adaptation to account for the semantics of the *forelem* loop. In this section, common optimizations that are carried out by optimizing compilers are introduced that will be referred to throughout this thesis. Optimizations that occur at the code generation level are not discussed in this thesis. These optimizations can be conducted on top of *forelem* transformations. Therefore, we will rely on common compiler tools, like for instance LLVM [63], to implement these backend transformations.

An important class of transformations are loop transformations, a number of which are now discussed. Loop blocking is often used to improve cache re-use of a loop nest by processing data in blocks. In [60] the influence of the stride of data access and the size of the blocks on cache efficiency is discussed. When Loop Blocking is used in conjunction with Loop Interchange, the data locality of a program can be further improved [35].

Using the Loop Interchange transformation it is possible to enable vectorization and/or parallelization by moving dependence cycles, and to increase cache re-use by improving the locality of the code. Loop Interchange can only be performed under certain conditions. Because the order in which statements inside loops are executed is changed by an interchange, the interchange of loops is valid only if the new order of statement execution preserves all dependencies of the old order. We will illustrate this using two examples from [104].

```
for (i = 1; i <= 100; i++)          for (i = 1; i <= 100; i++)
  for (j = 1; j <= 100; j++)          for (j = 1; j <= 100; j++)
    A[i][j+1] = A[i][j] * B[i][j]       A[i+1][j] = A[i][j+1] * B[i][j]
```

The loop on the left can be interchanged. This Loop Interchange will eliminate the dependency in level 2 (iterated by `j`) which makes it possible to vectorize the inner loop.

Contrary, the loop on the right cannot be interchanged. For example for $i = 3, j = 3$ the value `A[3][4]` is read and the value `A[4][3]` is written. The value `A[3][4]` is generated by the iteration $i = 2, j = 4$. In fact this statement at iteration $i = 3, j = 3$ has a dependency on the statement at iteration $i = 2, j = 4$. With the current nesting assuming standard execution order, the iteration $i = 2, j = 4$ will be executed before $i = 3, j = 3$. However, when the two loops are interchanged, or swapped, this will no longer be the case. The dependency has then been broken which makes this instance of Loop Interchange invalid.

To formally verify whether transformations are valid for a loop nest, a technique known as data-dependence analysis [58, 3, 5] is employed. The analysis re-

sults in data-dependence relations which reflect the constraints on the statement execution order.

Three classes of dependencies are usually distinguished: *(i)* a *true dependency*, where a value is first defined and then used (Read-After-Write), *(ii)* an *anti dependency*, when a value is read and then written (Write-After-Read), and *(iii)* an *output dependency* when a value is written and written again (Write-After-Write). A dependency between loop statements is either *loop independent*, indicating that the dependency holds for equal iteration vectors, or *loop carried*, meaning that the dependency holds for different iteration vectors of the loop.

When there exists a data dependency in the original loop that no longer exists in the loop-interchanged loop, this is said to be a loop-interchange preventing dependency. Observe that this was indeed the case in the second loop we discussed above.

A related analysis is def-use analysis [2, 50]. In this analysis statements are analyzed to see whether they are a definition (an assignment) or a use of a value. From this information definition-use and use-definitions chains can be set up for a variable in a basic block. This can be used to reason whether a variable is assigned a constant or whether an assigned (defined) variable is used at all, etc. Optimizations such as constant propagation and variable substitution use this analysis.

For a more thorough and formal treatment of these analysis techniques, we refer the reader to the cited literature as well as to [104] for a concise overview.

Two loops (at the same level if contained in a larger loop nest) can be merged into a single loop under certain conditions using the Loop Fusion transformation [52]. For now, we only consider serial loops. Consider the separated loops and the fused loop:

```
for (i = 0; i < 100; i++)
  A[i] = B[i] + C[i];

for (j = 0; j < 100; j++)
  D[j] = A[j] + X[j];
```

```
for (i = 0; i < 100; i++)
{
  A[i] = B[i] + C[i];
  D[i] = A[i] + X[i];
}
```

Loop Fusion is defined if the loops to be fused iterate the same iteration space and valid if there does not exist a dependency which prevents fusion of serial loops. Fusion is prevented if there is a dependency from a use in the second loop to a definition in the first loop for which $i > j$ holds true. When the loops would be fused, the value would be read before it is written. This can be verified by replacing the read of `A[j]` in the second loop with `A[j + 1]` and similarly in the fused loop.

Another transformation on loops is Loop Collapse. The Loop Collapse transformation is used to rewrite two levels of loops as one level of loop by using a one-dimensional representation of a two-dimensional array [100]. Loop Collapse is mostly used to enable vectorization and can only be applied on serial loops, that is, no loop-carried dependencies are present. Because two loop levels are collapsed into a single loop level, the vector length that can be used is increased [104]:

```
int A[100][100], B[100][100],
    C[100][100];

for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[i][j] * C[i][j]
```

```
int A[100][100], B[100][100],
    C[100][100];

for (i = 0; i < 10000; i++)
  A[0][i] = B[0][i] * C[0][i];
```

Note that in languages that support vector statements, such as Fortran, the resulting single-level loop can be written as a single statement.

Other examples of loop optimizations are loop unrolling, loop rerolling, loop fission and iteration space morphing. Zima [104] provides a comprehensive treatment of this kind of compiler optimizations. Finally, [95] discusses how loops that traverse a data structure using a pointer can be turned into counted loops operating on arrays that indirectly access the original data structure. This technique enables the application of existing loop optimization methods which are often not successful when applied on the original pointer-traversing loop.

Loop Invariant Code Motion is a kind of common subexpression elimination where statements which are invariant under the inner loop iteration variable can be moved to an outer loop or completely out of the loop nest. If the statement is, for example, a memory load, then the pressure on the memory bus can be significantly reduced. A simple example of the application of this optimization is:

```
for (i = 0; i < 100; i++)
{
  Y = 154;
  A[i] = B[i] + Y;
}
```

```
Y = 154;
for (i = 0; i < 100; i++)
{
  A[i] = B[i] + Y;
}
```

Vectorizing compilers have the capability to recognize reduction operations. Reduction operations are operations such as computing the sum or product of all elements of a vector, or determining the minimum or maximum element of a vector [77]. The compiler will replace a loop performing such an operation with a vector statement performing the same operation. The vector statement is implemented using specific vector instructions. Consider the following example, taken from [77], which computes the sum of a vector `A`:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  ASUM = ASUM + A(I)
END DO
```

```
A(1:N) = B(1:N) + C(1:N)
ASUM = ASUM + SUM(A(1:N))
```

The loop, written in Fortran, can be replaced with two vector statements. The `SUM` function used in the example returns the sum of the vector provided as argument.

Scalar Expansion is a transformation that is typically used to enable parallelization of loop nests. Consider the following loop:

```
for (k = 1; k <= N; k++)
  {
    tmp = A[k] + B[k];
```

```
    C[k] = tmp / 2;
  }
```

Due to the loop-carried anti-dependency of `tmp`, subsequent iterations cannot write to `tmp` before `tmp` has been used in the assignment to `C[k]`. This is solved by the Scalar Expansion transformation which expands the scalar `tmp` to a vector:

```
for (k = 1; k <= N; k++)
  {
    tmp[k] = A[k] + B[k];
    C[k] = tmp[k] / 2;
  }
```

Now that the loop-carried dependency has been broken, the loop can be parallelized.

In Global Forward Substitution, the right-hand side of an assignment statement is substituted into the right-hand side of other assignment statements [58]. This potentially eliminates flow dependencies, but also eliminates temporary variables in a subsequent dead code elimination phase. The well-known Constant Propagation optimization is considered a special case of Forward Substitution [77]. An example of Forward Substitution, adapted from [77], is:

```
NP = N + 1
```

```
for (i = 1; i < N - 1; i++)
{
  B[i] = A[NP] + B[i];
  A[i] = A[i] - 1
}
```

```
for (i = 1; i < N - 1; i++)
{
  B[i] = A[N + 1] + B[i];
  A[i] = A[i] - 1
}
```

In the code on the left side, there is a dependency between the two statements within the loop body. The compiler can typically not reason about the address of `A[NP]`, thus it is possible that `A[i]` overwrites the value used by `A[NP]`. The code on the right side is the result after Forward Substitution. Now, `NP` has been substituted with `N + 1`. From the expression the compiler can determine that the assignment to `A[i]` will never reach `A[N + 1]`, because the inclusive upper bound of the loop is `N - 2`. So, after Forward Substitution, the dependency no longer holds.

1.2 Database Systems

Part I of this thesis describes a global integrated optimization process for the optimization of web applications. The web applications that are used as a starting point, make use of a Database Management System (DBMS) for accessing persistent data. Within this optimization process, the application and data management codes are integrated with each other. This section briefly discusses work that is related to this integration process.

The purpose of a DBMS is to store data at a central location, concurrently accessible by multiple clients. It is the responsibility of the DBMS to safeguard the data, such that it does not become corrupted or inconsistent. To this extent, database systems implement the ACID properties: *atomicity*, *consistency*, *isolation* and *durability*. This ensures, respectively, that database transactions are always atomic; the database is always left in a consistent state; transactions that are running concurrently are fully isolated; and that committed data is guaranteed to be stored in a way such that it cannot be lost due to system crashes or power loss in the future. Clients can access this data by submitting *queries*, written in a declarative programming language such as SQL. A DBMS needs to translate these queries to a query plan, or execution plan, that specifies which actions need to be carried out and in which order, to process the query. During this translation stage transformations can be applied to the query plan to optimize the query execution, a process known as query optimization.

Work that is related to the global integrated optimization process as described in this thesis roughly touches two areas in the field of databases. The first area concerns the integration of the usage of database systems into imperative programming languages. The second area concerns applying optimizing compiler technology to query planning and optimization. In this section, both areas are explored in turn.

1.2.1 Integration of Database Systems Query Processing Into Imperative Programming Languages

Integration of query processing of database servers into programming languages is an area that has been under research for a long time. In this area, this often is referred to as the "impedance mismatch" [25, 65]. The impedance mismatch refers to the mismatch of elements of procedural programming languages and declarative database languages, such as procedural types versus database types, optimizations in procedural programming languages versus database query optimizations, concurrency versus transactions, etc. It is this mismatch that highlights the key problem in integrating usage of a DBMS in a programming language. Cook et al. [25] give a thorough review of the definition of this problem and approaches to solve it.

There are many examples of techniques that try to overcome this impedance mismatch. Most of these solutions focus on the integration of the database application programming interface (API) into programming languages and setting up a mapping between application code value types and database value types. Note that most of these techniques are tools that aid programmers to more effectively create database applications, and do not have as goal to support a integrated optimization process as is proposed in this thesis.

LINQ, for example described in [31], extends the programming language such that declarative queries can be naturally expressed. Programmers can express queries in LINQ without having to know how the query will be executed. Queries can be executed on a variety of data sets, for example on collections in main memory internal to the program and also on a DBMS. When a query expressed in LINQ is compiled, no code is generated to actually execute the query. This

is handled at run-time, when a query parse tree is built for the query. In case LINQ is used with a DBMS, LINQ will generate a set of SQL queries that are sent to the DBMS for execution. Given that the query parse tree is built at run-time, opportunities to extensively merge the query code with the application code at compile-time are not fully exploited.

Systems similar to LINQ also exist for example for the C++ language [39]. The focus is on solving the impedance mismatch and checking for correctness and security concerns (SQL injection attacks) at run-time. This is much like the static analysis of the correctness of SQL statements proposed by [20, 27]. In these papers, analyses are described to find security problems in the application code's usage of database APIs and to incorporate the time spent in the DBMS in the application profiling respectively. The latter also supports the rewriting of SQL queries, which is for example possible when it is detected that three columns are projected in the query but only two of those are used in the code. In such cases, the programmer can be alerted by the development environment. Also, an approach to static analysis of strings containing SQL fragments has been described, which results in compile-time warnings of problems in these fragments [96]. This analysis is based on the SQL grammar specification and the schemas of the target database.

Note that these systems solely concentrate on the facilitation of integration of the database API into the programming language realm. They do not concern the impedance mismatch in optimization as described in [25] and do not propose techniques for the explicit break down of layers between the application and database codes.

A different way to integrate database and application codes is to express database queries in the same imperative language as the application code. This was the approach taken by a specific class of programming languages known as Database Programming Languages (DBPLs). These languages are characterized by the fact that they include the ability to iterate through sets. In order to obtain good performance it is critical that DBPL compilers are extended to include database-style optimizations, such as join reordering. Initial work into such compile-time optimizations is described in [62]. The described transformations make standard transformation-based compilers capable of optimizing iterations over sets that correspond to joins. This work was later extended to include transformations that enable the parallelization of loops in DBPLs [61].

The Tycoon project aimed to replace special-purpose representations for queries, programs and scripts with a single expressive intermediate language [38]. This intermediate language, the Tycoon Machine Language, was based on continuation passing style. Continuation passing style is a functional programming construct wherein a function call has as last argument another function to call (named the continuation) once the called function has finished execution [89]. The language was used to move towards an integrated database language where user-defined code and query expressions are fully integrated. No final results or benchmark figures were published [25].

Contrary to expressing application and database codes in a common representation to perform integrated optimizations, it is also possible to devise transformations that operate on both the original application and database codes simulta-

neously. Such transformations have been researched by several groups. Holistic transformations for web applications are proposed in [66, 37]. The papers argue that tracking the relationship between application data and database data might yield advancements. It is exactly this relationship that we aim to exploit with the global integrated optimization methodology that is described in this thesis, however, we do not track the relationship, we rather eliminate this relationship by integrating the application and DBMS codes.

A similar holistic approach for Java code bases is described in [21], which motivates the approach by stating that rewrites of queries and programs are done independently by the database query optimizer and the programming language compiler. This independence leaves out many optimization opportunities. Their approach centers around a tool which aims to bridge this gap by performing holistic transformations on the program code and queries.

Cheung et al. describe a system, StatusQuo, to optimize the performance of database applications written in Java accessing a database through JDBC or Hibernate by considering both the application code as well as the queries [22]. Similar to us they state that the hard separation between the application and database code often results in applications with suboptimal performance. The system is capable of automatically partitioning the database application into a Java and SQL code, for optimal performance. To accomplish this, it may rewrite SQL into Java code, or vice versa. Whereas StatusQuo translates imperative code into a declarative form, our system translates declarative code into an imperative form and generates executable code from this imperative form.

The UltraLite system, described in [102], combines application and database logic together in one program. Given a program using embedded SQL for the UltraLite system, the query is compiled to C code which executes this query. This is done by sending the query to the host database server for parsing and optimization. The returned plan is used to generate the C code. The C code makes use of UltraLite run-time functions which implement SQL functionality. Concurrent execution of queries is supported by the run-time library. UltraLite is meant for usage on mobile devices with no hard disk and very little memory.

While the integration of the application and database logic may seem similar to what we are proposing, the integration limits itself to simply placing the application and database logic in a single executable. The generated C code does call functions in the run-time library, which diminishes the possibility to fully integrate the application and database codes at a code level. We propose a much finer grained integration of application and database codes, by intertwining these codes, which is not described in the cited patent.

A different way to perform optimizations that affect both the application code as well as the database queries consists of migrating this integration to the start of the application's design. GignoMDA is a framework to generate applications and database schemas for different programming platforms based on the Model Driven Architecture (MDA) approach [41]. The novelty of GignoMDA is that it promises to exploit cross-layer optimizations between the different layers of a database application in this framework. Typically, there is a presentation layer (e.g. a Web interface), a business logic layer and a persistence layer (e.g. the DBMS). By giving hints during the UML design process, for example hinting that a table

will mostly be used for read-only access, optimizations based on this hint can be carried out across all layers. The approach taken by GignoMDA only works on newly written applications using the MDA approach. This is different from our proposed code optimization backend, which works on existing codes and aims to do the optimization automatically rather than relying on hints.

1.2.2 Query Planning and Optimization

In query planning and optimization an execution plan is devised for a given query. Traditionally, the speed of disk I/O was the main bottleneck in query execution and thus query plans were traditionally optimized to minimize disk I/O. However, with the emergence of main-memory DBMSs such as MonetDB [16, 18, 17], the optimization objective has shifted from optimizing for minimal disk I/O to making best use of the available main memory bandwidth and exploitation of the CPU caches. Similar to optimizing compilers being equipped with techniques to improve caching reuse and utilization of memory bandwidth, query optimizers have to be equipped with such techniques as well.

The importance of optimizing for CPU cache re-use and vector processing was also demonstrated by the X100 query engine for MonetDB [18]. By mapping queries to *primitives*, simple C functions that apply a given operation on a given input vector in a tight loop, a one to two orders of magnitude performance improvement compared to existing DBMS technology was demonstrated. The sizes of the vectors to be used during the query processing are selected in such a way that they all fit in CPU cache. Optimizing compilers are responsible for compiling the primitives, written in C, into highly efficient code by application of aggressive loop pipelining and vectorization optimizations.

Another methodology is to architecture query optimization in such a way that use can be made of the transformations implemented in optimizing compilers. Recent research has explored the possibility of translating a query to an imperative code that can be processed and optimized by an optimizing compiler. For example, a strategy to transform entire queries to executable code is described in [57]. The technology, called "holistic query evaluation", works by transforming a query evaluation plan into source code, based on code templates, and compiling this into a shared library using an aggressively optimizing compiler. The shared library is then linked into the database server for processing. Although significant speedups over traditional and currently-emerging database systems are achieved, this approach does not include the integration of application and query evaluation codes. Instead, these codes remain separated because the query code is isolated in a shared library. Optimizing compiler technology is used to compile a translation of the query plan into C/C++ code through the use of code templates into efficient executable code.

In [73] a data-centric approach to query compilation is described. SQL queries are translated to relational algebra, which is optimized and from which LLVM assembly code is generated. The query in LLVM assembly code is then executed using the optimizing JIT compiler included with LLVM. The approach is data-centric in that the LLVM code is written such that data can be kept in CPU registers as long as possible for optimal performance. This is given more importance

than clearly maintaining the boundaries of relational operators. In fact, the relational operators are "blurred" when generating the code and the operators can be spread out over multiple code fragments. This technique results in very efficient query codes. However, note that just the query is taken into account during query compilation and the DBMS/application code split still exists.

DBToaster [1] is described as a novel query compilation framework for producing high performance compiled query executors that incrementally and continuously answer standing aggregate queries using in-memory views. DBToaster compiles queries into C++ code that incrementally maintain aggregate views at high update rates. The focus of DBToaster is on compiling queries to view maintenance code, contrary to the translation of entire queries which return the result of the query as you would normally receive it from a DBMS.

Compiler optimizations have also been used to take on the problem of multi-query optimization. In [9] an approach is demonstrated where queries are written as imperative loops, on which compiler optimization strategies are applied. The use of loop fusion, common subexpression elimination and dead code elimination is described. This work is tailored towards a certain class of analysis queries and not to generic queries. Furthermore, the loop fusion transformation described in the paper works by detecting multidimensional overlap. So, the strategy of loop fusion is used, but not an exact mapping of the traditional loop fusion optimization.

A different approach to multi-query optimization is described in [47], where optimization techniques are applied to the "algorithm-level" of a database program. In the algorithm-level, a query is represented as a sequence of algorithms, e.g. selection, join, that should be performed to compute the query results. The exact implementation of the algorithms is not made explicit at this level. As a consequence, knowledge is required about the implementation of algorithms that can appear in the representation by the optimizer in order to be able to carry out optimizations.

In [11], a "For-Loop Approach" is described to better handle aggregated subqueries that contain *where* clauses that overlap with the main query's *where* clause. By introducing a "for-loop operator" and "for-loop program", which are used together with relational algebra, the subqueries with overlapping *where* clauses can be integrated into the main query, eliminating redundant iterations over tables. The proposed "for-loop" operator and programs are meant to be an extension or tool to standard relational algebra.

As can be seen from this summary, many different approaches have been proposed to make DBMSs and especially query optimization more efficient. However, none of the approaches described above exploit (existing) compiler optimizations to their fullest extent.

1.3 Sparse Computations

In Part II of this thesis, the use of the *forelem* loop for the optimization of irregular applications will be explored. In particular, the utilization of the *forelem* intermediate for code and data structure generation of sparse matrix computations will be

discussed. Contrary to dense matrix computations, sparse matrix computations are irregular applications because specific data structures are used to store the matrix data instead of a regular two-dimensional array. From these specific data structures matrix elements with a zero value are omitted. An example of such a data structure is pointer-linked data structure in which elements that are located in the same row or column are linked to each other. Because these elements may not be placed in memory in consecutive order, the memory access may be very irregular. Irregular memory access in sparse computations is also caused by the selection of a data structure that does not store the matrix elements in the order in which they are accessed by the computation, also yielding ineffective use of the CPU cache next to irregular memory access.

There is a large body of literature on the optimization of sparse computations. More recent work covers overcoming the memory wall in modern CPUs using compression techniques [97, 56], the optimization of sparse matrix-vector multiplication on multicore platforms [98], the optimization for register reuse [44], and the implementation of matrix-vector multiplication on GPUs [14].

Many of the solutions described in the literature, such as specific algorithms and data structures are implemented in sparse algebra libraries. An extensive amount of work has been invested in designing several sparse libraries for different storage formats and computer architectures [14, 87, 81, 10]. While these libraries in general provide an adequate solution and are used frequently by code developers to generate optimal codes, use of these libraries is relying on predefined code. The libraries have fixed storage formats and especially when hybrid storage formats are needed, one cannot expect all different combinations to be pre-defined in the library.

Bik and Wijshoff described compiler techniques to automatically generate an implementation of a computation that operates on sparse matrix structures from an implementation of that computation expressed in terms of dense matrices that is supplied to the compiler [15]. User annotations about matrix statistics (e.g. its sparsity) or interactive user input is used to aid the compiler in selecting an efficient, pre-defined, sparse storage format for the matrices used in the computation.

Mateev et al. proposed a generic programming methodology to bridge the gap between algorithm implementation API and storage format API [68]. Algorithms are implemented as generic dense matrix programs, without considering a particular data storage format. The details of different, pre-defined, data structure formats are exposed using a low-level API. Their framework views sparse matrix formats are indexed-sequential access data structures and uses a restructuring technology based on relational algebra to convert a high-level algorithm into a data-centric implementation that exploits characteristics of the available sparse formats whenever possible. Matrices are considered as collections of tuples for the purpose of restructuring towards an existing sparse storage format. This restructuring process is covered in more depth in [55], which expresses the iterations, or tuples, that should executed as relational queries and finds a solution for these relational queries through the application of join reordering and determining suitable join algorithms to compute the joins.

Marker et al. described a method for the automatic parallelization and optimization of Dense Linear Algebra for distributed-memory computers called De-

sign by Transformation (DxT) [67]. Their method works by modeling algorithms in a data-flow graph. The graph contains nodes that represent redistribution operations or a LAPACK or BLAS function call. Optimization is carried out by applying graph transformations to find equivalent graphs that potentially exhibit better performance.

1.4 Contributions of Part I

The contributions of the first part of this thesis are developments towards a versatile intermediate representation for the optimization of codes and the development of a global, integrated, optimization framework for database applications. More specifically, the contributions of Part I of this thesis are:

1. An initial study of the cost of the overhead of the modular development methodologies for web applications that are in use today. This study shows that up to 90% of the instructions can be eliminated without affecting the final result, leading to substantial savings in energy consumption of web servers and a tremendous improvement of the performance of the web application. This work has been published in [84].

2. A tuple-based intermediate representation, the *forelem* loop, in which SQL queries can be naturally expressed in terms of loops governed by simple control. Extensions are described that allow nested and aggregate SQL queries to be represented.

3. The re-targeting of established compiler optimizations onto the *forelem* loops. The foundations of the intermediate representation and the re-targeted compiler optimizations have been published in [83].

4. Additional transformations for *forelem* loops and strategies for the optimization of database queries expressed in terms of *forelem* loops solely by using simple compiler optimizations. This optimization methodology results in executables codes that evaluate queries with a performance comparable to that of contemporary state-of-the-art database systems.

5. An automatic global integrated optimization process to reduce database applications to their essence. For two web applications it is shown that this automatic optimization process is very effective, on average eliminating 75% of the instructions and in specific cases up to 95% without affecting the execution and output of the application.

6. A trade-off analysis to support a decision process to determine whether it is beneficial to change the data access codes to be in-process (i.e. vertical integration) for a given sequence of queries. This analysis can be used by an automatic optimization process to determine for what parts of a web application it is beneficial to perform integral optimization, implying the construction of a local copy of the data. This work has been published in [85].

1.5 Contributions of Part II

The second part of this thesis contributes techniques to support automatic generation of code and data structures from a *forelem* representation of a problem in terms of tuples. More specifically, the contributions of Part II are:

1. Extensions to the *forelem* framework for the automatic generation of data storage formats from a representation of the code that operates on tuples. These extensions consist out of a materialization and concretization phase. In the case of sparse matrices, using these techniques established data storage formats, such as Jagged Diagonal Storage, can be automatically derived, that could up till now only be derived by hand. Part of this work has been published in [83].

2. A characterization of the search space consisting of many different variants of a sparse matrix computation represented in terms of tuples that can be automatically generated using the *forelem* framework. This characterization shows that by performing an exhaustive search through this search space, variants of the computation can be found that are in most cases faster than the implementations of these computations supplied by sparse algebra libraries, and at least on par in performance.

3. A further extension of the *forelem* framework, the *ready clause*, that allows dependencies between tuples to be naturally expressed. As such, generic irregular computations can be expressed in terms of tuples. By expressing dependencies as dependencies between tuples, it is trivial to deduce which operations on tuples can be executed at the same time. Preliminary experiments show that from an ordinary triangular solver code expressed in terms of a dense matrix, a highly parallel implementation is automatically derived that operates on sparse storage. This automatically derived implementation has a performance that is competitive to that of hand-optimized implementations. This work has been published in [86].

4. Initial work to support the expression of distributed *forelem* loops. By putting the optimization process in control of how *forelem* loops are distributed, optimal data decompositions and distributions can be determined. This extends the capability of the *forelem* framework to also optimize data composition and distribution in additional to the optimization data storage formats that are used locally. Part of this work has been published in [83].

1.6 Outline

This thesis is organized as follows. In Part I a framework is described for global, integrated, optimization of web applications. Chapter 2 presents an initial study of the cost of the overhead of the modular development methodologies for web applications that are in use today. It is shown that up to 90% of the instruction can be eliminated without affecting the final resulting, leading to substantial savings

in energy use of web servers and a tremendous improvement of the performance of the web application. This chapter has been published in [84].

Chapter 3 introduces the *forelem* loop, which is the main constituent of the *forelem* framework. A *forelem* loop specifies iteration of a subset of a multiset of tuples. SQL queries can be naturally expressed in terms of *forelem* loops. Transformations can be carried out on these loops to optimize the performance.

Chapter 4 extends the basic *forelem* framework introduced in Chapter 3 with syntax and transformations for handling nested SQL queries. In Chapter 5 further extensions are proposed for handling aggregate functions and group-by queries.

Chapter 6 describes how queries that are expressed in terms of *forelem* loops can be effectively optimized solely through the use of simple compiler optimizations.

Chapter 7 discusses how the global integration optimization process of database applications can be performed automatically for reducing database applications to their essence.

Chapter 8 describes a trade-off analysis that can be used by an automatic optimization process to determine for what parts of a web application it is beneficial to perform integral optimization, implying the construction of a local copy of the data, and for which parts this is not beneficial. This chapter has been published in [85].

Part II of this thesis explores the use of the techniques developed for the integral optimization of database applications in other application domains. Chapter 9 describes extensions to the *forelem* framework to support the automatic generation of data structures from a representation of a problem in terms of tuples. Parts of this chapter have been published in [83].

Chapter 10 characterizes the search space that consists of many different variants generated from an initial representation of a sparse matrix computation in *forelem* through the application of transformation. It is shown that by performing an exhaustive search through this search space, variants of the computation can be found that are in most cases faster than the implementations of these computations supplied by sparse algebra libraries.

Chapter 11 further extends the *forelem* framework with a *ready clause*. Using this clause, dependencies between tuples can be naturally expressed, resulting in a method that is especially suited for the automatic parallelization of irregular codes. This chapter has been published in [86].

Chapter 12 proposes a syntax for expressing and controlling distributed execution of *forelem* loops. It is described how this makes the *forelem* viable for the optimization of Big Data applications.

Finally, Chapter 13 summarizes the thesis and discusses future perspectives of the *forelem* framework.

PART I

Optimization of Database Applications

CHAPTER 2

Quantifying Instruction-Count Overhead of Web applications

2.1 Introduction

Applications that have seen a steady rise in ubiquity in the last 10 years are web applications. In many data centers web applications are hosted that provide much of the World-Wide Web's content. Web applications are often built from several readily available components to speed up development, such as web development frameworks and database management systems (DBMSs). This modularity allows for rapid prototyping, development and deployment. The World-Wide Web has heavily benefited from this modular approach.

However, it is well known that modularity does not come for free. In the last decade, the increasing energy consumption of data centers has already drawn the attention of governments. The US Environment Protection Agency (EPA) reported on the energy efficiency of data centers in a 2007 report [93]. Their study says energy consumption of US data centers had doubled in the period from late 2000 to 2006. Based on this trend, they projected another doubling in electricity use for the period from 2006 to 2011. This would mean a quadrupling in electricity use by data centers in about 10 years time.

Contrary to the EPA prediction, a 2011 report by [46] claims electricity by US data centers increased by 36% instead of doubling from 2005 to 2010. This is significantly lower than predicted by the EPA report. Koomey attributes this to a lower server installed base than predicted earlier, caused by the 2008 financial crises and further improvements in server virtualization. Nevertheless, the energy consumption is still increasing and while the increase was only 36% in the US, according to the Koomey report the increase amounted to 56% worldwide.

The server installed base is an important metric, because each installed server adds up to the amount of electricity used, not only due to the energy used by the server itself but also because cooling capacity has to be increased. The EPA report makes many recommendations to reduce electricity use in data centers.

Many of these recommendations seek for solutions in hardware. Better power management can make servers more energy efficient. The server installed base can be reduced by making better use of virtualization to consolidate servers. Recommendations are also made to develop tools and techniques to make software more efficient by making better use of parallelization and to avoid excess code. However, these recommendations are not as concrete as those for hardware improvements.

An interesting observation in the EPA report is that those responsible for selecting and purchasing computer equipment are not the same as those responsible for power and cooling infrastructure. The latter typically pay the electricity bills. This leads to a split incentive, because those who could buy more energy efficient hardware have little incentive to do so. We believe a similar split incentive exists with software developers. Software developers are not in charge of obtaining the necessary computing equipment in data centers and also are not aware of the electricity bills. Therefore, software developers have very little incentive to further optimize the software to reduce energy consumption. The fact that software optimization is a very diligent and costly task does not help.

Many statistics have been collected on data center cost and performance. For example [24] describes several metrics that are used in data center management to optimize performance and drive costs down. Many of these metrics concentrate around MIPS or the number of servers and processors. Mainframe size is expressed in MIPS, and cost can be expressed in spending (on hardware, software, personnel, etc.) per MIPS. Statistics on Intel-based UNIX and Windows operating environments are expressed in number of servers and processors. In such statistics a distinction is made between *installed MIPS* versus *used MIPS*. These numbers should not be too far apart, because server capacity staying idle is neither cost nor energy efficient. However, as soon as a server is no longer idle, the work performed is counted as *used MIPS*. Of such used MIPS it is not investigated whether these MIPS did useful work or were mainly overhead. In other words, no distinction is made between *essential MIPS* and *non-essential MIPS*, with non-essential MIPS being accrued from the cost of the usage of rapid development frameworks and software modularity.

In this chapter, we address the quantification of energy usage through non-essential instruction (MIPS) count overhead. To be able to effectively target efforts to reduce energy consumption of web applications, the sources of overhead must be quantified. An instruction-level quantification of overhead in web applications does not exist to our knowledge. We investigate three existing and representative web applications and show that in this manner accurate measurements on spilled energy usage can be obtained. In our opinion, this will result in an incentive for data centers to prioritize the need for software overhead reduction instead of only improving hardware energy efficiency.

In Section 2.2 we give an overview of how non-essential MIPS are determined in web applications. Section 2.3 describes the experimental setup and the web applications used. The sources of overhead are quantified in Section 2.4. Section 2.5 links the results of the experiments to the expected reduction in energy consumption. Section 2.6 discusses work related to this chapter. Finally, Section 2.7 lists our conclusions.

2.2 Determination of non-essential MIPS

A number of categories of overhead, or non-essential MIPS, in web applications that make use of the PHP language and a MySQL DBMS can be identified. The instruction count of non-essential MIPS for each category is obtained by counting instructions in the original program code and in the program code with the overhead source removed. The difference in instruction count is the number of instructions for the corresponding overhead source.

The first category of overhead we will consider is overhead caused by development frameworks for web applications. Many applications are written with the use of a framework which dramatically shortens development time and increases code re-use. One of the applications we survey in this chapter has been developed using the CakePHP framework. The CakePHP simplifies development of web applications by performing most of the work serving web requests and abstracting away the low-level data access through a DBMS. Frameworks like this affect performance by, for example, performing unnecessary iterations and copies of results sets, or even by executing queries of which the results are not used at all.

The second category of overhead is the PHP language itself. Due to PHP's nature as a script language, there is a start-up overhead due to parsing and interpretation of the source files and the potential to thoroughly optimize the code in a similar way to compiled languages is removed.

As a third category, we consider the current modular design of DBMSs, which has as result that a single DBMS instance can be easily used for a variety of applications. A downside of this modularity is that when an application has a request to retrieve data, this request has to go out of process. Depending on the architecture of the website, the request is either served by a DBMS running on the same server as the web server executing the PHP code or the request is sent to a remote DBMS host. Overhead that is incurred can include: context switching overhead, network and protocol overhead and data copying overhead.

The fourth category of overhead is caused by DBMS APIs implemented in shared libraries. The library sends PHP queries to the DBMS and retrieves the results. As a result of this architecture details on the data accesses are shielded off and also overhead is introduced by iterating result sets at least twice: once when the DBMS builds up the result set and sends this to the application program and once in the application program itself when the results are iterated.

2.3 Experimental Setup

In this chapter we quantify three web applications: *Discus*, *RUBBoS* [74] and *RUBiS* [75]. Although it can be questioned whether these three applications are representative of typical web applications running in data centers, we are convinced that at least these benchmarks can be used to create an initial quantification of the overhead. In our experiments, we have focused on read-only workloads.

Discus

Discus is an in-house developed student administration system. The system is based on CakePHP development framework, version 1.2.0. To quantify the overhead for full page generations, all forms of caching (e.g. query or page caching) in CakePHP have been disabled. We focus on two particular page loads: the students index and the individual student view. All experiments have been carried out with three different data sets: small (25.000 tuples), medium (0.5 million tuples) and large (10 million tuples).

RUBBoS

The RUBBoS benchmark was developed by a collaboration between RICE University and INRIA and models a typical bulletin board system or news website with possibility to post comments [74]. We have used the PHP-version of RUBBoS. To quantify the overhead in different typical pages served by RUBBoS, we looked at the page generation times of the following pages: *StoriesOfTheDay*, *BrowseStoriesByCategory*, *ViewStory*, *ViewComment*. We believe these pages are typical for the workload that characterizes a news story website. As data set we have used the data set that is made available at the RUBBoS website.

RUBiS

The RUBiS benchmark [75] models a simple auction website and is similar to the RUBBoS benchmark as it has been developed by the same collaboration. For our tests with RUBiS we have used the same methodology as with RUBBoS. The PHP-version of RUBiS was used and the following pages were picked: *ViewBidHistory*, *ViewItem*, *ViewUserInfo* and *SearchItemsByCategory*. The data set that is available from the RUBiS website has been used as data set in our experiments.

The experiments have been carried out on an Intel Core 2 Quad CPU (Q9450) clocked at 2.66 GHz with 4 GB of RAM. The software installation consists out of Ubuntu 10.04.3 LTS (64-bit), which comes with Apache 2.2.14, PHP 5.3.2 and MySQL 5.1.41. No extraordinary changes were made to the configuration files for Apache and MySQL, except that in MySQL we have disabled query caching to be able to consistently quantify the cost for executing the necessary queries. In the experiments we obtained two metrics. First, we obtained the page generation time which we define as the difference between the time the first useful line of the PHP code started execution until the time the last line of code is executed. Secondly, we acquire the number of instructions executed by the processor to generate the page by reading out the `INST_RETIRED` hardware performance counter of the CPU.

2.4 Quantification

In this section, we will quantify the overhead for each of the categories described in Section 2.2. To quantify the overhead induced by the PHP programming lan-

guage we have compared the performance of the three code bases to the code bases compiled to native executables using the HipHop for PHP project [42]. This project is developed by Facebook and is a compiler that translates PHP source code to C++ source code, which is linked against a HipHop runtime that contains implementations of PHP built-in functions and data types. Both the Apache HTTP server and the PHP module are replaced by the HipHop-generated executable.

In Figures 2.1, 2.2 and 2.3 the difference in page generation time between Apache and the HipHop-compiled executables is shown as "PHP overhead". Figures 2.4, 2.5 and 2.6 depict the number of instructions executed to generate the page. For the Discus code base we observe that roughly $1/3$ to $2/3$ of the page generation time can be attributed to PHP overhead. A similar overhead is found in the figures depicting the number of instructions

The RUBBoS and RUBiS benchmarks do not show an improvement in page generation time when the HipHop-version is tested. However, the instruction count does noticeably decrease. Compared to Discus, the RUBBoS and RUBiS source code is quite straightforward and it is possible that aggressive compiler optimizations do not have much effect. Although less instructions are retired, it is plausible that instead more time is spent waiting for (network) I/O.

To quantify the overhead of development frameworks, we have focused on one of the main sources of overhead in the CakePHP framework, which is the data access interface or in particular the automatic generation of SQL queries. The difference between the code bases with and without automatic query generation is displayed in Figures 2.1 and 2.4 as "CakePHP gen. overhead". For most cases, the overhead of automatic query generation ranges from $1/12$ to $1/5$ of the total page generation time. This is significant if one considers the total page generation time in the order of seconds and the fact that there is no technical obstacle to make the queries static after the development of the application. CakePHP employs caching to get around this overhead (which was disabled to uncover this overhead). However, caching does not help if similar queries are often executed with different parameters. We have not done similar experiments for RUBBoS and RUBiS, because these code bases use static queries instead of a rapid development framework.

We made the application code compute the results of the SQL queries instead of sending these queries to a different DBMS process, to quantify the overhead of using common DBMS architecture. This was done by modifying the HipHop-compiled Discus, RUBBoS and RUBiS code bases by replacing calls to MySQL API with code that performs the requested query. For example, calls to the function `mysql_query` were replaced with a complete code that performs the requested query and fills an array with the result tuples. Algorithmically seen, the query is performed in exactly the same manner as it would have been performed by MySQL. Using the Embedded MySQL Server Library [70], it is also possible to perform SQL queries within the client process without contacting a remote DBMS. This approach does not address the DBMS API overhead as described in Section 2.2, because the generic MySQL API function calls remain in use, shielding off the data access from the application code. Furthermore, our objective is to obtain a minimum amount of instructions required for processing queries, to emphasize the cost of using a generic, modular DBMS.

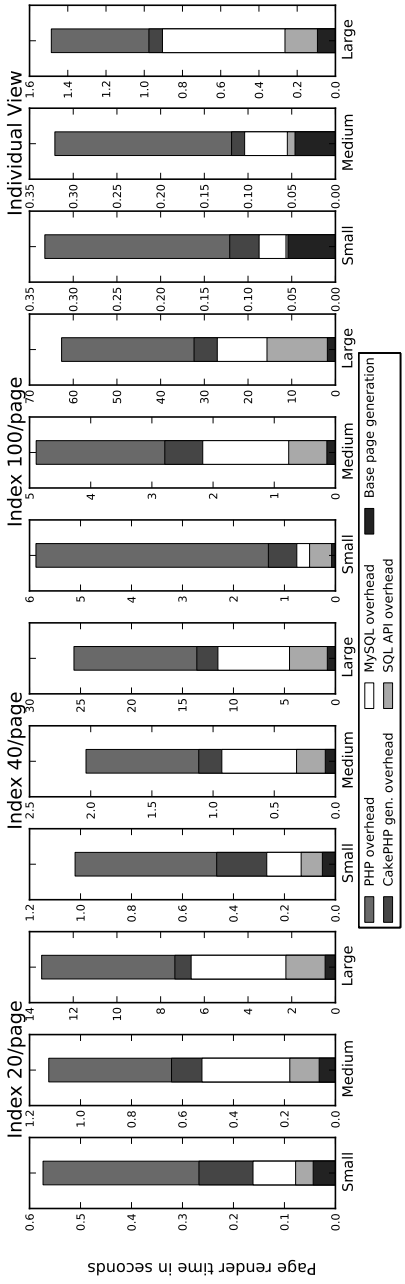


Figure 2.1: Page generation time in seconds for different pages and data set sizes from Discus. Each category (e.g. "Index 20/page") contains up to 10 different page loads over which the average over 5 runs is taken.

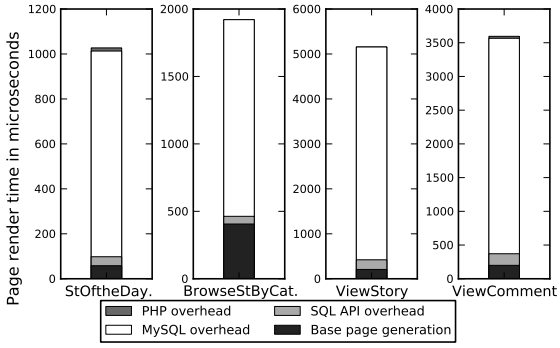


Figure 2.2: Page generation time in microseconds for different pages from the RUBBoS benchmark. Times displayed are averages of 10 runs on a warmed-up server.

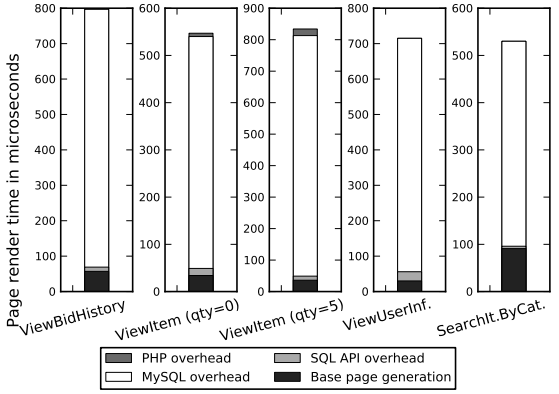


Figure 2.3: Page generation time in microseconds for different pages from the RUBiS benchmark. Times displayed are averages of 10 runs on a warmed-up server.

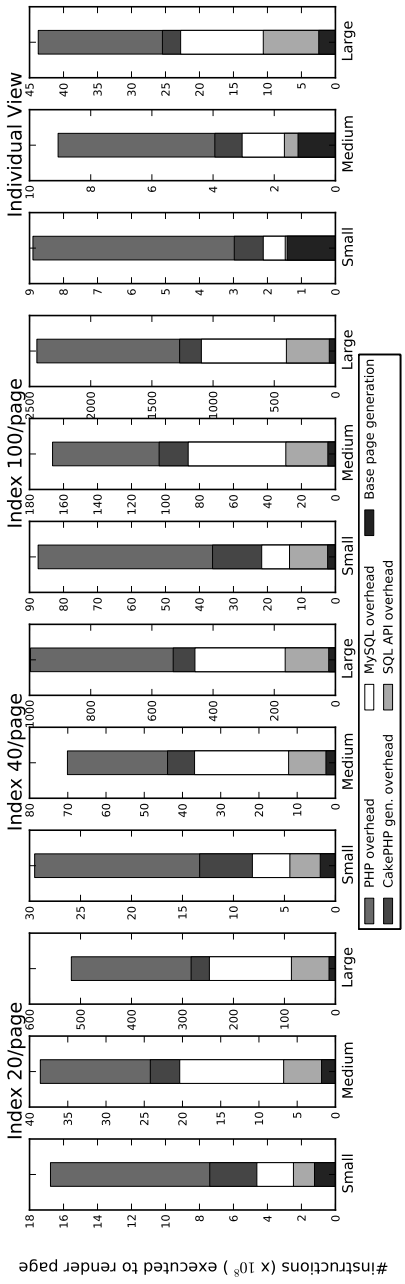


Figure 2.4: Instruction count in 10^8s of instructions for different pages and data set sizes from Discus.

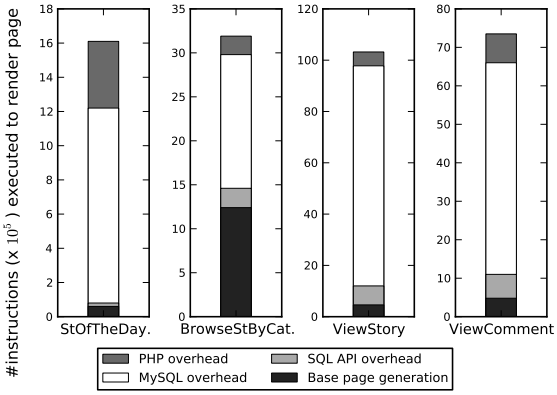


Figure 2.5: Instruction count in $10^5$of instructions for different pages from the RUBBoS benchmark.

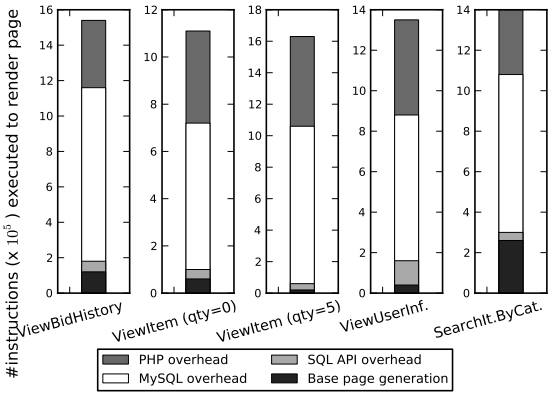


Figure 2.6: Instruction count in 10^5s of instructions for different pages from the RUBiS benchmark.

The time results are displayed in Figures 2.1, 2.2 and 2.3. Compared to the original execution time of the Apache-version of the Discus application, the MySQL overhead accounts for roughly 8% to 40% of the execution time. If we compare the MySQL overhead to the execution time of the original HipHop-compiled version however, the MySQL overhead accounts for roughly 20% to 60% of the execution time. These larger numbers are also reflected in the RUBBoS and RUBiS results, where the MySQL overhead is estimated to be around 72% to 90% of the page generation time. Put differently, elimination of the MySQL server improved the page generation times for the RUBBoS and RUBiS applications by about a factor 10. We again see similarly sized reductions in instruction count in Figures 2.4, 2.5 and 2.6.

Once the SQL queries have been expanded to code inside the application code, several sources of overhead of DBMS APIs become apparent. It is now possible to integrate the application code with the query computation code in a much more optimal fashion. In the majority of the cases the loops that iterate over the result set can be merged into the loops that perform the actual queries and thus build the result sets, saving an iteration of the result set.

In Figure 2.1 we observe that it is possible to achieve at least a factor 2 speedup in most cases compared to the HipHop-version of Discus. For the Index 100/page with the large data set case, a factor 10 speedup is obtained. Although the overhead of the DBMS API does not appear that significant compared to the total execution time, it does prove to be of significance when put in the perspective of the execution time of the application when the usage of MySQL has been removed. The same reasoning holds true when the results of the RUBBoS and RUBiS applications are analyzed, displayed in Figures 2.2 and 2.3 respectively. When compared to the total execution time of the derivative of the application with the MySQL overhead removed, removing the SQL API overhead results in speedups close to a factor of 2 in half of the cases.

2.5 Energy Consumption

From the results collected we can deduce that in general the time spent per instruction stays in the same order of magnitude both when non-essential (overhead) instructions are removed as well as when the problem size is increased. Overall, there is an approximately linear relation between the page generation time and the amount of instructions executed to generate a page. For the applications that have been surveyed, removal of non-essential instructions has an immediate approximately linear impact on performance.

Tables 2.1 and 2.2 show the number of non-essential instructions that are executed for each essential instruction for the Discus and RUBBoS/RUBiS experiments respectively. The overhead varies greatly from page to page. In the majority of cases however, the overhead is significant: more than 20 non-essential instructions are executed for each essential instruction. An overhead of a factor 20.

In the Discus experiments we observe a trend that as the data set size increases, the overhead increases as well. Put differently, the overhead expands when the

		Ratio
20/page	**Small**	14.92
	Medium	23.26
	Large	41.67
40/page	**Small**	22.72
	Medium	30.30
	Large	45.45
100/page	**Small**	43.48
	Medium	38.46
	Large	47.62
Indiv. View	**Small**	4.69
	Medium	6.17
	Large	18.52

Table 2.1: Displayed is the ratio of non-essential instructions executed for each essential instruction.

	Ratio
RUBBoS, BrowseStByCat.	1.56
RUBBoS, ViewComments.	14.29
RUBBoS, ViewStory.	21.28
RUBBoS, StoriesOfTheDay.	25.64
RUBiS, SearchItByCat.	4.4
RUBiS, ViewBidHistory	11.83
RUBiS, ViewItem-qty-0	17.54
RUBiS, ViewItem-qty-5	80.64
RUBiS, ViewUserInfo	32.26

Table 2.2: Displayed is the ratio of non-essential instructions executed for each essential instruction.

same code is ran on a larger data set. It is very well possible that this increase in overhead is caused by the data reformatting done in the CakePHP framework. This happens when the result set received from the DBMS is formatted into an array that can be used for further processing by CakePHP framework objects. The fact that the overhead of the MySQL and SQL API categories increases linearly with the data size and thus stays constant per row also indicates that the cause of this increase in overhead should be sought in the CakePHP framework. From this result it can be expected that in this case the reduction in non-essential instructions will be larger as the data set size increases.

We showed a significant reduction in MIPS to be executed by a factor of 10. However, if we look at the ratio of time per essential instruction versus the average time per non-essential instruction (instruction time delay) for the web applications we have surveyed (Table 2.3), we observe that for most cases the average time per essential instruction is larger than the average time per non-essential instruction. A possible explanation for this is that the majority of essential instructions are expected to be carrying out memory access or disk I/O. As a result, when estimating

	ITD
RUBBoS, BrowseStByCat.	0.42
RUBBoS, ViewComments.	0.84
RUBBoS, ViewStory.	0.90
RUBBoS, StoriesOfTheDay.	1.55
RUBiS, SearchItByCat.	0.91
RUBiS, ViewBidHistory	0.91
RUBiS, ViewItem-qty-0	1.16
RUBiS, ViewItem-qty-5	3.63
RUBiS, ViewUserInfo	1.43
Discus 20/page, Medium	1.42
Discus 40/page, Medium	1.30
Discus 100/page, Medium	1.15
Discus Indiv. View, Medium	1.03

Table 2.3: The Instruction Time Delay (ITD) is shown, which is the ratio of the average time per essential instruction against the average time per non-essential instruction. The latter is the weighted average of the time per instruction of the different overhead categories.

energy reduction this has to be taken into account, see below. In case of RUBBoS and RUBiS, this trend is not always visible. Possibly, this is because the time spent by the essential instructions is quite small, due to small transfers of data, so that these do not weigh up to the time spent by the non-essential instructions.

Table 2.4 displays the instruction time delay for Discus experiments with varying data set sizes. For all cases, the time delay increases as the data set size increases. Or, as non-essential instructions are eliminated the essential instructions responsible for fetching the data from memory will take more time to execute. The memory wall becomes more exposed as overhead is removed. Recall from Table 2.1 that the number of overhead instructions increases with the data set size, however, the instruction time delay increases as well and this will counteract the increase in overhead.

When we consider the majority of the essential instructions to be carrying out memory access or disk I/O, we have to consider the cost of such data accesses in our conservative estimates of impact on energy usage. We use the component peak powers and actual peak power, which is 60% of the advertised peak power, of a typical server described by [33]. In the very worst case, essential instructions are the most expensive in energy usage and non-essential the cheapest, the removal of overhead would then only remove the cheap instructions. If we take the typical machine's actual peak power usage of just the CPU and memory for the essential instructions and the idle power usage (estimated at 45% of actual peak power) for the CPU and memory for the non-essential instructions, we obtain a ratio of approximately $69.6W : 31.3W$ or a factor 2.22. Let dP be this factor 2.22, dT be the average time per essential instruction versus the average time per non-essential instruction (or time delay), reported in Table 2.4, and R be the ratio of non-essential versus essential instructions, reported in Table 2.1. Then, we can use the formula

		ITD
20/page	**Small**	1.24
	Medium	1.42
	Large	1.50
40/page	**Small**	1.18
	Medium	1.30
	Large	1.45
100/page	**Small**	0.55
	Medium	1.15
	Large	1.46
Indiv. View	**Small**	0.91
	Medium	1.03
	Large	1.25

Table 2.4: The Instruction Time Delay (ITD) is shown, similar to Table 2.3, but for Discus experiments with varying data set sizes.

$$\left(1 - \frac{dT \times dP}{R+1}\right) \times 100\%$$

to get a worst-case estimate of the energy saving. Completing this formula for $dP = 2.22, dT = 1.45, R = 45.45$ gives an estimated energy saving of 93.1%. A little more realistic, we can estimate the energy saving considering the entire machine by taking the idle power usage of the entire machine for non-essential instructions and a peak energy usage of 90% of actual peak power usage of the entire machine for the essential instructions. This results in a dP of 2.0. When we complete the formula with this dP, we get a slightly higher energy saving estimate of 93.8%.

Our expected energy saving is based on the observation that essential instructions are more expensive than non-essential instructions, because essential instructions are more frequently instructions that access data in memory and disk I/O. Benchmarks with typical servers show that power usage of such servers is between 60% and 80% of actual peak power [33]. If we consider essential instructions to use 80% of actual peak power and non-essential instructions to use 60%, we obtain a dP of 1.33. With the same parameters for dT and R, we estimate an energy saving of 95.8%.

If we do not make a distinction in energy used per instruction for essential and non-essential instructions, we can compare the obtained numbers with the estimated energy saving This is reflected in the following formula:

$$\left(1 - \frac{dT}{R+1}\right) \times 100\%$$

Completing for $dT = 1.45, R = 45.45$ results in an estimated energy reduction of 96.9%. This is higher than the expected saving, because we do not consider essential instructions to be more expensive.

Table 2.5 lists the estimated energy savings for the various Discus experiments. We note that the energy savings increase corresponds to an increase in data set

		Estimated Energy Saving
20/page	**Small**	89.6%
	Medium	92.2%
	Large	95.3%
40/page	**Small**	93.4%
	Medium	94.5%
	Large	95.8%
100/page	**Small**	98.4%
	Medium	96.1%
	Large	96.0%
Indiv. View	**Small**	78.7%
	Medium	80.9%
	Large	91.5%

Table 2.5: Estimated Energy Saving using the expected $dP = 1.33$, dT obtained from Table 2.4 and R from Table 2.1

	Estimated Energy Saving
RUBBoS, BrowseStByCat.	78.2%
RUBBoS, ViewComments.	95.0%
RUBBoS, ViewStory.	94.6%
RUBBoS, StoriesOfTheDay.	92.3%
RUBiS, SearchItByCat.	77.6%
RUBiS, ViewBidHistory	90.6%
RUBiS, ViewItem-qty-0	91.7%
RUBiS, ViewItem-qty-5	94.1%
RUBiS, ViewUserInfo	94.3%

Table 2.6: Estimated Energy Saving using the expected $dP = 1.33$, dT obtained from Table 2.3. Values for R not shown due to space constraints.

size. Even though the instruction time delay is increasing with the size of the data sets, we still observe an increase in energy saving. We conclude that the increase in overhead instructions superfluously counteracts the instruction time delay. This is due to the significant size of the overhead ratios. Consider for example Discus Index 20/page; where the number of overhead instructions doubles for each expansion in data set size (Table 2.1), the instruction time delay only increases with a factor 1.05 to 1.15 (Table 2.4). Results obtained using the same formulas for the RUBBoS and RUBiS experiments are shown in Table 2.6.

In conclusion, the estimated lower bound on energy saving we have found in the experiments performed with Discus amounts to 71% for the Individual View experiment with the small data set. This result correlates well with the page generation times displayed in Figure 2.1, where approximately $3/4$ of the page generation time is eliminated when the overhead is removed. Similar results are found in the results of the RUBBoS and RUBiS benchmarks, with a lower bound of 77.6%.

We believe this is significant, especially when considering that the CPU and memory consume almost half of the energy used by the entire server.

2.6 Related Work

Research has been done into the performance characteristics of collaborative Web and Web 2.0 applications that emerged in the last decade. [90] examined a collaborative Web application, where most content is generated by the application's users, and show that there is a fundamental difference between collaborative applications and real-world benchmarks. [71] compare Web 2.0 workloads to traditional workloads. Due to the use of JavaScript and AJAX at the receiving end, many more small requests for data are made. The research shows that Web 2.0 applications have more "data-centric behavior" that results in higher HTTP request rates and more data cache misses. The quantification as presented in this chapter does not attempt to characterize workloads, instead it presents a low-level quantification of where time is spent in program codes executing web applications.

In [6] a tool, WAIT, is introduced to look for bottlenecks in enterprise-class, multi-tier deployments of Web applications. Their tool does not look for hot spots in an application's profile, but rather analyzes the causes of idle time. It is argued that idle time in multi-tier systems indicates program code blocking on an operation to be completed.

For our quantification we have used the HipHop for PHP project [42] to translate PHP source code to native executables. Similarly, Phalanger compiles PHP source code to the Microsoft Intermediate Language (MSIL), which is the bytecode used by the .NET platform. The effect of PHP performance on web applications has thus been noted in the past and we believe the existence of these projects is an indication that PHP overhead is a valid concern for large PHP code bases.

We have argued that development frameworks for web applications bring about overhead by, for example, the generality of such frameworks and the abstract data access interface. In [101] the authors argue that large-scale Java applications using layers of third-party frameworks suffer from excessive inefficiencies that can no longer be optimized by (JIT) compilers. The main cause of such inefficiencies is the creation of and copying of data between many temporary objects necessary to perform simple method calls. One reason why this problem is not easily targeted is the absence of clear hotspots. In [101] a technique is introduced called "copy profiling" that can generate copy graphs during program execution to expose areas of common causes of what the authors refer to as "bloat".

One of the goals of the DBMS overhead elimination is to move both the query loop as well as the result set processing code into the same address space, so that they can be optimized together as we have done to estimate the DBMS API overhead. Some approaches to optimize both the database codes and the applications codes do already exist, for example, the work on holistic transformations for web applications proposed by [66] and [37]. These papers argue that tracking the relationship between application data and database data is a tool that might yield advancements. Note that by eliminated the DBMS overhead we do exploit this relationship, but rather by eliminating the relationship by integrating application

and DBMS codes than by tracking this relationship. A similar approach for holistic transformations for database applications written in Java is described by [21].

2.7 Conclusions

In this chapter we described and quantified several sources of overhead in three web applications. This quantification indicates that there is a tremendous potential for optimization of web applications. Of the total number of instructions executed to generate the web pages in the investigated applications, close to 90% of the instructions can be eliminated. Removal of these non-essential instructions has an approximately linear relationship with the decrease in page generation time. This results in faster response times as well as significantly reduced energy usage. We have seen a lower bound on energy savings of approximately 70% for all experiments performed in this study.

Considering the simplicity of the RUBBoS and RUBiS code bases, we believe that the estimates shown for these applications are a lower bound on the performance that can be gained. Still, the results are impressive. The more complex Discus application shows that performance increases between one and two orders of magnitude are a possibility, solely by reducing the number of non-essential instructions that are executed.

CHAPTER 3

Specification of the Forelem Intermediate Representation

The previous chapter described an initial study on overhead in web applications showing that 90% of the instructions executed to generate web pages are non-essential; in other words, these can be eliminated without affecting the final result. This could result in a saving of energy consumption by computer hardware in data centers between 70% and 95%. A large part of the 90% reduction of the amount of executed instructions is due to integration of the code to evaluate database queries into the application code. By the combined integration of application codes and the database requests, it becomes possible to optimize applications by optimizing compiler technology after the division between application and DBMS codes has been eliminated.

To help this integration, we propose a methodology to efficiently express database queries in terms of an imperative language and thus allowing for integration of the application code with the code performing the evaluation of the database query. Within this methodology, queries are expressed in such a way that full integration in the work flow of common optimizing compilers is achieved. This makes it possible to unleash the full power of optimizing compilers on the combination of application and database codes.

Current development environments and frameworks to develop, for example, Java-based database applications or PHP-based web applications, have been serving programmers very well. They enabled programmers to rapidly develop and deploy complex web applications. Without these technologies, the World-Wide Web would not have made such a large advancement as it did in the last decade. So, whatever change we are proposing to improve DBMS performance, this development methodology should be kept in place.

To eliminate the observed overhead, our aim is to develop a code optimization backend, or global integrated optimization process, that is able to take an existing database or web application and automatically breaks down the layers that incur overhead. This code optimization backend will co-exist with contemporary development methods and frameworks for web applications. An application is

developed and tested as usual, but before extensive deployment in a data center, the code is passed through the code optimization backend to eliminate as much overhead as possible. This way, we continue to take advantage of the available software development tools which enhance programmer productivity and combine this with a code optimization backend that significantly improves the performance of the application and reduces the energy consumed by the hardware which runs the application.

In order to realize the code optimization backend, a methodology is needed to efficiently express database queries in terms of an imperative language and that allows for integration of application code with the code performing the evaluation of the database query. Within this methodology, the database queries must be expressed in such a way which allows for full integration in the work flow of common optimizing compilers. In this chapter, we introduce such a method: *forelem* loop nests. *forelem* loop nests are designed to integrate DBMS queries in a normal optimizing compiler work flow, and also support many database-style optimizations such as the use of various kinds of join algorithms. It is important to note that *forelem* loop nests will only be used by the code optimization backend and it is explicitly not our intention to present *forelem* as a new programming methodology to write database applications.

This chapter demonstrates that simple SQL queries can be expressed in terms of *forelem* loops. In Chapters 4 and 5 extensions are proposed so that nested queries, aggregate functions and group-by queries can be expressed. When a query is expressed in the *forelem* framework, the complexities of query evaluation are encapsulated and what remains is a collection of simple loops. As will be described in this chapter, such loop nests are very well suited for optimization by optimizing compilers and a number of transformations will be described that can be applied on *forelem* loops nests, such as loop merge, loop interchange and loop collapse. After describing the new notation and transformations, the use of these transformations will be demonstrated by their application on a real-world code example. This example will show the power of the usage of *forelem* loop nests and the described transformations.

We believe that this methodology might be a solution to the "impedance mismatch" in optimization as described in the first chapter [25]. The mismatch illuminates the fact that there is a mismatch in how application codes are optimized compared to how database statements are optimized. We are convinced that *forelem* loop nests eliminate this mismatch, even though it is claimed that explicit looping structures give a particular implementation of the query that limits the range of transformations and evaluation choices [65]. Maier says that in databases, iteration is encapsulated so that the system can pick the iteration form. Despite that *forelem* loop nests appear to be explicit looping structures, the power is in the usage of "index sets". Because of the use of "index sets", iteration is still encapsulated and the iteration form is picked during optimization. Which iterations and which index sets are needed will emerge from the optimization phase performed by the code optimization backend. Based on this information, the most efficient way to compute and use the index set is determined, which is equivalent to picking the iteration form.

3.1 The forelem Loop Nest

In this section the basics of *forelem* loop nests are introduced. Each *forelem* loop iterates a (multi)set of tuples. Tuples in these multisets are accessible with subscripts, like ordinary arrays. The subscripts that are accessed through an "index set" that is associated with the multiset.

The *forelem* loop will be described through the use of simple SQL queries. Throughout the discussion, we will use queries inspired by the "Sailors" database described in [82]. This database consists out of the following table schemas:

Sailors : $\underline{\text{sid}}, \text{sname}, \text{rating}, \text{age}$
Boats : $\underline{\text{bid}}, \text{bname}, \text{color}$
Reserves : $\underline{\text{sid}}, \underline{\text{bid}}, \text{day}$

Let us consider a first query:

```
SELECT S.sname
FROM Sailors S
WHERE S.rating = 7
```

The query is expressed in relational algebra as $\pi_{sname}(\sigma_{rating=7}(S))$ and is executed by performing the selection $rating = 7$ on table S (Sailors) and storing $sname$ from matching tuples into the result set. A C code to evaluate this query could look as follows:

```
for (i = 0; i < len(Sailors); i++)
  {
    if (Sailors[i].rating == 7)
      add_to_result(Sailors[i].sname)
  }
```

In this code fragment, the *for* loop iterates the full `S` table, the *if*-statement selects matching tuples, corresponding to the σ operator, and the `add_to_result` function performs the task of the π operator.

The main problem with this code fragment is that the looping structure is explicit, which already gives a particular implementation of the query and this limits the range of transformations and evaluation choices [65]. It is apparent that a full iteration over the Sailors table is to be done, to check the rating of each Sailor. This explicit looping structure excludes the possibility to, for example, exploit an index on the rating values. Additionally, more complex query constructs, such as *distinct* and *group by*, require more complicated code to represent using only standard C language constructs such as *for* and *if*. The downside of this is that more complicated code is harder to apply transformations to and hides the actual problem at hand.

Ideally, only those rows are iterated for which the condition $rating = 7$ holds true. This is similar to what an index on the column rating would accomplish. One way to accomplish this is to move the definition of these conditions into the loop control structure, in our case the *for* statement. As a result, the explicit *if*

statements are eliminated, which paves the way for the application of a larger range of optimizations. The above query loop written using *forelem* looks like the following:

```
forelem (i; i ∈ pS.rating[7])
  ℛ = ℛ ∪ (S[i].sname)
```

This code fragment is read as follows: with `i`, iterate over each index into table `S` for which `rating == 7` holds true. For these i, we append a tuple containing the value of `sname` for index `i` into table `S` to the result set $\mathscr{R}$.

Even though the *forelem* loop appears to be very similar to a *foreach* loop that exists in many common programming languages, there is one distinguishing feature. This concerns the notation `pS.rating[7]`. This denotes that a set of indices into table `S` will be returned for which the `rating` field equals 7. This is similar to an *index set* as is commonly used in DBMSs, and we will also use this term to refer to the sets of indices we define here. The fact that an index set contains indices is indicated by the prefix *p*, from pointer. Note that the order in which the indices appear in the index set is not defined. From this follows that the exact semantics of how the table `S` will be iterated are not set in stone at this point. Contrary to the original *for* loop, the *forelem* loop does not have an explicit looping structure and does not impose a particular implementation of the query. Because of this, we are not limited in the range of transformations and iteration schemes we can apply. Index sets are the essence of *forelem* loop nests as they encapsulate iteration and simplify the query loop code so that aggressive compiler optimizations can be successfully applied.

Before proceeding, some further notation and terminology is introduced first. In this chapter, the focus is on expression SQL queries as *forelem* loops. As a result, the *forelem* loops will iterate database tables. A database table contains tuples, that contain data for one or more columns. In a database table a tuple is not necessarily unique, therefore a database table is a *multiset*. A multiset is a set in which elements may occur more than once, furthermore, the order of items in the multiset is irrelevant. So, in general, *forelem* loops specify iteration of (a subset of) a multiset of tuples.

Let `D` be a multiset representing a database table. `D` can be indexed with a subscript `i` to get access to a tuple, or row, in `D`: `D[i]`. A specific field of a row can be accessed with `D[i].field` where `field` is a valid field of `D`. Without subscript, an entire column is selected resulting in a multiset containing all values of that column: `D.field`.

In *forelem* codes that have been generated from SQL statements, the loop body often outputs tuples to a temporary or result set. Temporary sets are generally named $\mathscr{T}_1, \mathscr{T}_2, ..., \mathscr{T}_n$ and result sets (or output relations) $\mathscr{R}_1, \mathscr{R}_2, ..., \mathscr{R}_n$. These temporary tables and result sets are both multisets. The semantics that apply to multisets representing database tables apply to temporary tables as well.

An *index set* is a set containing subscripts `i` $\in \mathbb{N}$ into an array. Since each array subscript is typically processed once per iteration of the array, these subscripts are stored in a regular set. Index sets are named after the array they refer to, prefixed with "p".

pD represents the index set of all subscripts into a database table D: $\forall t \in D : \exists! i \in pD : D[i] = t$. D can also be a temporary table $\mathscr{T}_n$. All rows of D are visited if all members of pD have been used to subscript D. Random access by subscript into pD is not possible, instead all accesses are done using the $\in$ operator. $i \in pD$ stores the current index into i and advances pD to the next entry in the index set.

The part of the table that is selected using an index set can be narrowed down by specifying conditions. For example, the index set denoted by pD.field[k] returns only those subscripts into D for which field has value k. This can also be expressed as follows:

$$pD.field[k] \equiv \{i \mid i \in pD \wedge D[i].field == k\}$$

Similarly, the index set from the example query can be expressed as follows:

$$pS.rating[7] \equiv \{i \mid i \in pS \wedge S[i].rating == 7\}$$

When a match on multiple fields is required, the single column name is replaced with a tuple of column names:

$$pD.(field1, field2)[(k_1, k_2)] \equiv \{i \mid i \in pD \wedge D[i].field1 == k_1 \wedge D[i].field2 == k_2\}$$

Instead of a constant value, the values k_n can also be a reference to a value from another table. To use such a reference, the table, subscript into the table and field name must be specified, e.g.: D[i].field. This notation is especially suited for expressing equi-joins.

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

This query finds the name of all sailors who have reserved the boat with id 103. It contains an equi-join on the sid fields from the Sailors and Reserves tables. Expressed in *forelem*, we obtain:

```
forelem (i; i ∈ pR.bid[103])
  forelem (j; j ∈ pS.sid[R[i].sid])
    ℛ = ℛ ∪ (S[j].sname)
```

If a value should not be tested for equality but rather for greater or less than, a different notation is used. Instead of a single value an interval is written:

$$pD.field[(-\infty, k)] \equiv \{i \mid i \in pD \wedge D[i].field < k\}$$

Even though the notation implies a single index set, the interval is represented as the union of the individual index sets:

$$pD.field[(-\infty, k)] \equiv \cup_{i=-\infty}^{k-1} pD.field[i]$$

Note that when dealing with bounds of infinity, iterating over each possible index set with field = i is not useful if there are no subscripts into the table for a

specific i. However, *forelem* is only used as a representation during optimization and in the resulting final code, the final set might have been created already.

Regular set operations on index sets are possible, but only on index sets that relate to the same database table. Index sets contain subscripts into a specific table and therefore it is not possible to, for example, union index sets relating to different tables. Often, the union operation will be of use. For example, given two index sets on a different value of a field, say "color", the union of these index sets yields all subscripts into the database table for which color is either value.

3.2 Relationship Between SQL and forelem

In this section, we briefly sketch how SQL statements are translated into *forelem* loop nests. SQL statements with an arbitrary number of joins can be written as a *forelem* loop nest while preserving correctness of the results. This is because both the SQL statement and the corresponding *forelem* loop nest set up the same Cartesian product. This fact will be used to reason about the correctness of the translation and the conditions under which transformations can be applied on the *forelem* loops.

Consider the following query performing a join:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

which is expressed in relational algebra as $\pi_{sname}(\sigma_{S.sid=R.sid \wedge R.bid=103}(S \times R))$, or more commonly using the join operator: $\pi_{sname}(\sigma_{R.bid=103}(S \bowtie_{S.sid=R.sid} R))$. Theoretically, a join is performed by first setting up the Cartesian product over S and R and secondly selecting tuples which match the given conditions. We can write the first relational algebra expression as a *forelem* loop nest. The part $S \times R$ can be written as follows:

```
  forelem (i; i ∈ pS)
    forelem (j; j ∈ pR)
S1    ℛ = ℛ ∪ (S[i].*, R[j].*)
```

where S[i].* denotes all fields of table S at subscript i. In the result tuple all fields of S are suffixed with ^{S}. This loop nest sets up the Cartesian product $S \times R$ at statement S_1, which stores the Cartesian product in $\mathscr{R}$. After executing the loop nest, $\mathscr{R}$ is equivalent to what would be produced by the relational algebra expression $S \times R$.

The selection operator σ is implemented by making a pass over the result table and only storing matching tuples in a new result table. Of the matching tuples we only store the requested fields to implement the π operator.

```
  forelem (i; i ∈ pℛ)
    if (ℛ[i].sid^S == ℛ[i].sid^R && ℛ[i].bid^R == 103)
S1    ℛ2 = ℛ2 ∪ (ℛ[i].sname^S)
```

By application of the Temporary Table Reduction transformation that will be described in Section 4.4.2, both loops can be merged into one:

```
  forelem (i; i ∈ pS)
    forelem (j; j ∈ pR)
S1    if (S[i].sid == R[j].sid && R[j].bid == 103)
S2      ℛ = ℛ ∪ (S[i].sname)
```

In general, we say that a perfectly nested *forelem* loop nest of the following form:

```
 forelem (i1; i1 ∈ pT1)
   forelem (i2; i2 ∈ pT2)
     ...
       forelem (in; in ∈ pTn)
S1        ℛ = ℛ ∪ (T1[i1].field, ..., Tn[in].field)
```

sets up a Cartesian product of the tables $\mathtt{T}_1, \mathtt{T}_2, ..., \mathtt{T}_n$ at statement S_1. The Cartesian product, or rather the part of the Cartesian product that is accessed, must be preserved under any transformation for the query to yield correct results.

3.3 Transformations on forelem Loop Nests

forelem loop nests were devised such that common loop transformations could be applied to the resulting code. A number of transformations that can be applied to *forelem* loop nests are discussed in this section. These transformations are based on existing optimizing compiler techniques, such as these discussed in Chapter 1 (Section 1.1), and have been tailored for usage with *forelem* loop nests.

Generally, compiler optimizations are governed by data dependence analysis [58, 3, 5, 104]. The analysis results in data-dependence relations which reflect the constraints on the statement execution order. These constraints determine whether a given transformation can be applied without affecting the correctness of a program.

A related analysis is def-use analysis [2, 50]. In this analysis statements are analyzed to see whether they are a definition (an assignment) or a use of a value. This analysis is used to find unused variables, or to infer the current value of a variable by looking at preceding definitions of the variable in the def-use chain.

3.3.1 Loop Invariant Code Motion

Loop Invariant Code Motion is a kind of common subexpression elimination where statements which are invariant under the loop's iteration variable can be moved to an outer loop or completely out of the loop nest. Within the *forelem* framework this transformation is generally used to move condition testing of array fields to outer loops to prune the iteration space, or to inner loops so that further loop transformations will be enabled. For example:

```
forelem (i; i ∈ pX)
  forelem (j; j ∈ pY)
```

```
    if (X[i].field2 == value && Y[j].field2 == X[i].field1)
      ℛ = ℛ ∪ (Y[j].field1)
```

compares the value `X[i].field2` with a constant `value`. The reference `X[i].field2` is invariant under the inner loop, so can be moved to the outer loop. Fully moving the condition test out of the loop nest is not possible, because the array reference is variant under the outermost loop. The result is:

```
forelem (i; i ∈ pX)
  if (X[i].field2 == value)
    forelem (j; j ∈ pY)
      if (Y[j].field2 == X[i].field1)
        ℛ = ℛ ∪ (Y[j].field1)
```

Similarly, statements can be moved to the innermost loop, to enable the application of loop transformations, such as Loop Interchange.

3.3.2 Loop Interchange

The Loop Interchange transformation is derived from the common loop interchange transformation applied by optimizing compilers discussed in Chapter 1 and reorders the nesting of loops in a loop nest. To perform this transformation, Loop Invariant Code Motion is used to move the conditions to the inner loop before the loop nest is reordered and back to the outermost loop after the reordering.

The standard Loop Interchange transformation changes the order in which the statements in the loop are executed. This transformation is only valid if the new execution order preserves all dependencies of the original execution order [104]. Commonly, data-dependence analysis [58, 3, 5] is employed to formally verify whether the data-dependence relations are preserved across loop transformations. In general, only certain loop-carried dependencies can prevent application of Loop Interchange. A *forelem* loop does not specify a particular execution order and therefore loop-carried dependencies cannot exist. As a consequence, interchanges of loops in a perfect loop nest are always valid.

Loop-carried dependencies are only caused by dependencies of the loop bounds of inner loops on outer loop iteration counters. In this case, Loop Invariant Code Motion is first used to move the conditions to the inner loop before the loop nest is reordered and back to the outermost loop after the reordering. This way, Loop Interchange is applied to a perfectly nested loop nest.

Within the *forelem* framework the Loop Interchange transformation is used to reorder loops such that as many conditions as possible are tested in the outermost loop to prune the search space. As an example, consider the following loop nest over tables `X` and `Y` with a result table $\mathscr{R}$:

```
forelem (j; j ∈ pY)
  forelem (i; i ∈ pX.(field1, field2)[(Y[j].field2, value)])
    ℛ = ℛ ∪ (Y[j].field1)
```

First, the conditions of all loops are written as *if*-statement and moved to the inner loop nest using Loop Invariant Code Motion:

```
forelem (j; j ∈ pY)
  forelem (i; i ∈ pX)
    if (X[i].field1 == Y[j].field2 && X[i].field1 == value)
      ℛ = ℛ ∪ (Y[j].field1)
```

After the preparatory step, the *forelem* loop nest is in the perfectly nested form and generates the cross product of tables `X` and `Y` at the *if*-statement. The only dependency in this loop is the true dependency on $\mathscr{R}$ in consecutive iterations. On $\mathscr{R}$ the $\cup$ operator is used to indicate that tuples are being added to a (result) set. In this code fragment, no sort order is imposed on the result set, so the order in which the tuples appear in $\mathscr{R}$ does not matter (as long as the tuples are correct). This breaks the true dependency. Given that there are no other dependencies, the loops can be interchanged freely which does not change the contents of the generated cross product:

```
forelem (i; i ∈ pX)
  forelem (j; j ∈ pY)
    if (X[i].field1 == Y[j].field2 && X[i].field1 == value)
      ℛ = ℛ ∪ (Y[j].field1)
```

Next, Loop Invariant Code Motion is applied to move conditions to the outermost loops:

```
forelem (i; i ∈ pX)
  if (X[i].field2 == value)
    forelem (j; j ∈ pY)
      if (Y[j].field2 == X[i].field1)
        ℛ = ℛ ∪ (Y[j].field1)
```

As a final step, the conditions are moved from the *if*-statements into the index sets in the *forelem* loop nest:

```
forelem (i; i ∈ pX.field2[value])
  forelem (j; j ∈ pY.field2[X[i].field1])
    ℛ = ℛ ∪ (Y[j].field1)
```

Note, that as a result of this transformation, the comparison with `value` is performed in the outermost loop, which effectively prunes the iteration space.

3.3.3 Loop Fusion

Loop Fusion [52] is a traditional compiler optimization that can be readily applied to *forelem* loops. The transformation can, under certain conditions, merge two loops (at the same level if contained in a larger loop nest) into a single loop. Application of Loop Fusion is only prohibited by certain loop-carried dependencies. Such loop-carried dependencies do not exist in *forelem* loops. Therefore, Loop Fusion can be applied on two adjacent *forelem* loops if the iteration spaces of the two loops are equal. This is the case if the index sets for both loops refer to the same table and contain the same set of subscripts into these tables. After Loop Fusion has been applied, the bodies of both loops are then executed for the same set of subscripts into the same array. For example:

```
forelem (i; i ∈ pTable1)
  𝓡1 = 𝓡1 ∪ (Table1[i].field1)
forelem (i; i ∈ pTable1)
  𝓡2 = 𝓡2 ∪ (Table1[i].field2)
```

can be rewritten into the following:

```
forelem (i; i ∈ pTable1)
{
  𝓡1 = 𝓡1 ∪ (Table1[i].field1)
  𝓡2 = 𝓡2 ∪ (Table1[i].field2)
}
```

Note that *forelem* loops generally only access the array being iterated using the subscript of the current iteration. E.g., an access into an array always has the form i and not $i + 2$ or similar. As a consequence, a condition preventing Loop Fusion from being applied will in general not occur.

3.3.4 Loop Merge

A typical query in a database application has the following structure when expressed using *forelem* loops:

```
forelem (i; i ∈ pS.rating[7])
  𝓡 = 𝓡 ∪ (S[i].sname)
while (row ∈ 𝓡)
  print(row.sname);
```

The *forelem* loop is a loop producing result tuples, the *while* loop is a loop consuming these result tuples. Both loops enumerate the same iteration space, namely the set of all matching tuples. In this particular example, the temporary storage $\mathscr{R}$ is not required and can be eliminated. To eliminate the usage of temporary storage, the loops have to be merged, similar to how loops enumerating the same iteration space can be fused using Loop Fusion under certain conditions:

```
forelem (i; i ∈ pS.rating[7])
{
  row = S[i].sname;
  print(row.sname);
}
```

We will refer to this transformation as the Loop Merge transformation. Note that as a further optimization, the references to the result row can be rewritten to be references immediately into the database table.

Sometimes a preparatory transformation has to be done on the code before it is possible to perform the Loop Merge transformation. Consider the following code fragment:

```
forelem (i; i ∈ pS.rating[7])
  ℛ = ℛ ∪ (S[i].sname)
if (is_empty(ℛ))
  print("There are no matches.");
else
{
  while (row ∈ ℛ)
    print(row.sname);
}
```

To make it possible to apply the Loop Merge transformation, the *else*-clause has to be eliminated. We can reason that when the *if*-condition is true, the body of the *while* loop will never be executed. The `is_empty` function that is called in the *if*-condition is under our control and we can ascertain that this will not introduce side effects. This also holds true for the condition in the *while* statement. We can now place the *while* loop before the *if*-statement and eliminate the *else* clause:

```
forelem (i; i ∈ pS.rating[7])
  ℛ = ℛ ∪ (S[i].sname)
while (row ∈ ℛ)
  print(row.sname);
if (is_empty(ℛ))
  print("There are no matches.");
```

As a result, it is possible to apply Loop Merge.

More preparatory transformations are required in order to cover a wide range of database applications. Within the limitations of this thesis, they will not further be discussed.

3.3.5 Loop Collapse

With the Loop Collapse transformation, two *forelem* loops are collapsed into a single loop. In conjunction with this, the two tables iterated by these loops are merged into a new table. This is inspired by the original Loop Collapse transformation used in optimizing compilers which rewrites two levels of loops as one level of loop by using a one-dimensional representation of two-dimensional arrays [100]. Loop Collapse is a vectorization transformation and can only be applied on serial loops, that is, no loop-carried dependencies are present.

An sample scenario where Loop Collapse might pay off is when a second query is executed for each result tuple generated by the main query. In terms of the *forelem* intermediate representation, this second query is executed from within the loop body of the main query, as will be demonstrated in an example. After the transformation both the main and second queries can be satisfied by an iteration over a single table. Secondly, this transformation can be used to search for a potentially more effective table layout to compute the given query.

Consider the following loop nest over tables `X` and `Y` with a result table $\mathscr{R}$:

```
forelem (i; i ∈ pX)
  forelem (j; j ∈ pY.field2[X[i].field1])
    ℛ = ℛ ∪ (Y[j].field1)
```

Similar to the Loop Interchange transformation, we first write the conditions of all loops as *if*-statements and move these into the inner loop using Loop Invariant Code Motion:

```
forelem (i; i ∈ pX)
  forelem (j; j ∈ pY)
    if (Y[j].field2 == X[i].field1)
      ℛ = ℛ ∪ (Y[j].field1)
```

We note that the *forelem* loop nest is now in the perfectly nested form and generates the cross product of tables X and Y at the *if*-statement. The loop is serial in that there are no loop-carried dependencies and due to the use of the $\cup$ operator and the fact that no sort order is imposed no dependencies are enforced on $\mathscr{R}$.

The two loops are now collapsed as follows. First, a new table X$\times$Y is created by taking the cross product of X and Y. Secondly, the two loops are replaced with a single loop which iterates over the new table. The conditions and code to append a result are rewritten to use the new table.

```
forelem (k; k ∈ pX×Y)
  if (X×Y[k].field2^Y == X×Y[k].field1^X)
    ℛ = ℛ ∪ (X×Y[k].field1^Y)
```

The cross product X$\times$Y that has been created in the course of the transformation is equal to the cross product that is generated by the relational algebra expression X$\times$Y. Hence the *forelem* loop iterates over the same relation as the selection operator. Exactly the same cross product is generated at the *if*-statement compared to the non-collapse loops.

At last, conditions are moved into the *forelem* statement again:

```
forelem (k; k ∈ pX×Y.field2^Y[X×Y.field1^X])
  ℛ = ℛ ∪ (X×Y[k].field1^Y)
```

In this step, new syntax is introduced. Note that the value to match for the condition on field2$^{\text{Y}}$ is field1$^{\text{X}}$. When no table and no subscript specifying a row in the table are explicitly given, then the same table and row are used from which the field that is being compared to is obtained. In this case, we are comparing the value to field field2$^{\text{Y}}$ in table pX$\times$Y, so field1$^{\text{X}}$ is taken from table pX$\times$Y as well. Both of the fields are retrieved from the same row, in this case both from the row at subscript k.

So far, we have considered a perfectly nested loop. With additional analysis it is also possible to transform imperfect loop nests into a form such that Loop Collapse can be applied. Let us consider the following loop nest, for which we in light of this discussion assume that $\mathscr{R}_2$ is never empty at statement S_4:

```
    forelem (i; i ∈ pItems)
    {
      userId = Items[i].userId
S1    ℛ2 = ∅
      forelem (j; j ∈ pUsers.id[userId])
S3      ℛ2 = ℛ2 ∪ (Users[j].name)
S4    ℛ = ℛ ∪ (r2 ∈ ℛ2)
    }
```

This construction was inspired by a case from the RUBiS benchmark [75], which is further discussed in Section 3.4. Note that statement S_4 fetches a single tuple from $\mathscr{R}_2$ which is subsequently stored in result set $\mathscr{R}$.

The loop nest differs from the perfectly nested loop nest because the main result $\mathscr{R}$ is assigned at S_4, which is outside of the inner loop body and secondly because a second result set $\mathscr{R}_2$ is used. At S_2 the entire Cartesian product is generated by the loop nest, however, the full product is not stored in a result set. At S_3, only fields are stored from tuples where `Users.id` and `Item.userId` match. Due to S_4, only the first of these matches makes its way into the result set $\mathscr{R}$.

We can detect this more formally by applying def-use analysis on the loop nest. We assume a modified form of def-use analysis, which can interpret *forelem* loops and its index sets. Def-use analysis will indicate that after running the inner loop $\mathscr{R}_2$ will either contain $\emptyset$ (which we will in light of this discussion ignore), or one or more user names generated by S_3. Additionally, we know from the analysis that only the first item of $\mathscr{R}_2$ is used (S_4). For the inner loop this means it is sufficient to only perform the first iteration and to eliminate the remainder of the loop:

```
    forelem (i; i ∈ pItems)
    {
      userId = Items[i].userId
      j = j ∈ pUsers.id[userId]
S4    ℛ = ℛ ∪ (Users[j].name)
    }
```

In this loop, the execution flow of the original loop becomes more obvious. For each item, the username of the first user which matches the Item's `userID` is fetched.

With this knowledge, we will now transform the loop nest into a form such that Loop Collapse can be applied. Given a table `Users` with fields `id` and `name`, we add a column `idMask` such that `idMask` is set to 1 for each occurrence of a row in `Users` that makes it into $\mathscr{R}$. In this case, `idMask` is 1 for each first occurrence of a value in `id` and 0 for each following occurrence of that value.

Observe that when the column `id` solely contains unique values, or when it has been marked as a key in the original table schema, all values in `idMask` are 1. When we use the new column in the loop nest, we obtain:

```
    forelem (i; i ∈ pItems)
    {
S1    ℛ2 = ∅
```

```
      forelem (j; j ∈ pUsers.(id,idMask)[(userId,1)])
S3        ℛ2 = ℛ2 ∪ (Users[j].name)
S4    ℛ = ℛ ∪ (r2 ∈ ℛ2)
    }
```

Due to the use of the mask, we are now assured that S_3 will only generate a single tuple for a matching user. We can merge statements S_3 and S_4 and eliminate the usage of $\mathscr{R}_2$:

```
    forelem (i; i ∈ pItems)
      forelem (j; j ∈ pUsers.(id,idMask)[(userId,1)])
S4      ℛ = ℛ ∪ (Users[j].name)
```

The loop is now in perfectly nested form and suitable to be transformed using the defined transformations.

3.3.6 Reverse Loop Collapse

Reverse Loop Collapse is exactly the reverse operation of the Loop Collapse transformation. Of a specified *forelem* loop, the accessed table is split into two tables and the loop is replaced with two loops, each iterating one of the two newly created tables. How the fields of the large table will be divided over the two new tables, or how the table will be split, has to be specified. This information will be giving by the code optimization backend that is driving the application of the different transformations.

The Reverse Loop Collapse transformation can be applied after application of several Loop Collapse and Loop Interchange transformations. Due to reorderings of the loops in the loop nest, it is likely that different tables will be chosen to be split, compared to the tables which were combined (collapsed). This will potentially lead to different schemas for the database tables which allow the query to be computed more efficiently.

During optimization, the code optimization backend plays a central role in determining how loop nests are reordered, split and combined. When guiding the optimization, the backend will take into account all queries of a given application code. The intention is to arrive at database table schemas which make the application code as a whole run more efficiently and not to optimize for one specific query of that application code.

3.3.7 Horizontal Iteration Space Reduction

The aim of horizontal iteration space reduction is to remove unused fields from a table's schema. Let `T` be a table with fields `field1`, `field2`, `field3` and `field4`, `C` a list of condition fields `C` $\subset$ (`field1` `field2`) and `V` a list of values. Consider the loop nest:

```
forelem (k; k ∈ pT.C[V])
  ℛ = ℛ ∪ T[k].field1 + T[k].field2
```

We define a new table $\texttt{T}' \subseteq \texttt{T}$ with fields `field1`, `field2` and replace the use of `T` with `T'` in the loop:

```
forelem (k; k ∈ pT'.C[V])
  ℛ = ℛ ∪ T'[k].field1 + T'[k].field2
```

The loop now iterates the table `T'` which does not contain the fields `field3` and `field4`.

3.3.8 Vertical Iteration Space Reduction

The Vertical Iteration Space Reduction transformation is primarily used after the application of the Loop Collapse transformation. Recall that Loop Collapse may introduce conditions that test whether two fields of the same row as equal. In Vertical Iteration Space Reduction, rows for which such conditions do not hold are eliminated from the table. This elimination is valid because these rows are never visited in the inner loop. Two different cases are distinguished: `T.field1[field2]` and `T.field[k]`.

Case 1. `T.field1[field2]`

On a table `T`, let the condition be `T.field1[field2]`. This notation implies table `T` contains both fields name `field1` and `field2`. All rows for which `T.field1 != T.field2` holds true are removed from the table. In the resulting table `T'`, $\forall \texttt{t} \in \texttt{T}' : \texttt{t.field1} = \texttt{t.field2}$, so that either `field1` or `field2` can be removed. Let `T` be a table with fields `field1` ... `fieldn` and consider the loop nest:

```
forelem (k; k ∈ pT.field1[field2])
  ℛ = ℛ ∪ T[k].field3
```

A new table T', with all fields of T except `field2`, is defined as follows:

$$\texttt{T}' = \{\texttt{t} \mid \texttt{t} \in \texttt{T} \wedge \texttt{t.field1} = \texttt{t.field2}\}$$

and replaces `T` in the loop nest, additionally all uses of `field2` are replaced with `field1`:

```
forelem (k; k ∈ pT')
  ℛ = ℛ ∪ T'[k].field3
```

Note that in this specific case `field1` is no longer used after the transformation and can be removed by applying horizontal iteration space reduction.

Case 2. `T.field[k]`

Case 2 is similar to Case 1, however since we do not reduce an equality between two fields in Case 2, no field is removed by the reduction operation.

On a table `T`, let the condition be `T.field[k]` All rows for which `T.field != k` holds true are removed from the table. In the resulting table `T'`, $\forall \texttt{t} \in \texttt{T}' : \texttt{t.field} = \texttt{k}$.

Let `T` be a table with fields `field1` ... `fieldn` and consider the loop nest:

```
forelem (i; i ∈ pT.field[k])
  ℛ = ℛ ∪ T[i].field3
```

A new table `T'` is defined with the same fields as `T` as follows:

$$\mathtt{T}' = \{\mathtt{t} \mid \mathtt{t} \in \mathtt{T} \wedge \mathtt{t.field} = \mathtt{k}\}$$

Note that in this specific case `field` is no longer used after the transformation and can be removed by applying horizontal iteration space reduction.

This reduction can also be applied if instead of a constant `k`, an interval is given. All rows for which the complement of the interval holds true are removed from the table. For example, for an interval $(-\infty, 10)$ all rows for which the field matches the complemented interval $[10, \infty)$ are removed from the table. This results in the table `T'` with $\forall \mathtt{t} \in \mathtt{T}' : \mathtt{t.field} < 10$.

3.3.9 Table Reduction Operators

In many database operations first an expanded result table is generated after which it is reduced to using just a single subset of columns and/or rows. More specifically, this happens when explicit Cartesian products are computed to perform table joins, for instance in the Loop Collapse transformation. Note that in our transformation framework, we have implicit transformations which immediately reduce the generated Cartesian product to a set of reasonable size. These reductions are defined as Horizontal and Vertical Iteration Space Reduction and are driven by the conditions and selected columns of the query. In fact, because the Cartesian product only is referred to in intermediate codes when transformations are taking place, the full product is usually never instantiated.

We also define explicit table reduction operators in our framework. Whereas implicit table reduction operators do not affect the final result set of a query, explicit table reduction operators do. The expansion of queries in our framework into loop-based programs potentially enables the use of def-use analysis to detect explicit reductions of these expanded tables later on in the code. The following program fragment demonstrates two examples of such explicit reductions, one column based and one row based:

```
forelem (i; i ∈ pS.rating[7])
  ℛ = ℛ ∪ (S[i].*)
while (row ∈ ℛ)
  if (row.age > 18)
    print(row.sname);
```

We consider the program fragment before applying Loop Merge, so that the individual loops performing the query `SELECT * FROM Sailors WHERE rating = 7` and the loops processing the result set are clearly visible. Firstly, we observe that only rows with `age > 18` are being further processed and other rows are discarded. This allows us to apply row-based explicit table reduction to eliminate all rows with `age <= 18` from the result set $\mathscr{R}$. Secondly, we observe that the field *sid* is not used in the program fragment, although it is fetched from the table due to the * operator. Using column-based explicit table reduction, we further reduce the size of the result set by no longer storing *sid* in result rows.

3.3.10 Combined transformations

The real power of the transformations presented in this section is unleashed when they are combined. Let us consider the following code fragment, which is generated from a simple database application code performing the query

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

and enumerating the result set to output the sname field for each row. This results in the following intermediate code:

```
forelem (i; i ∈ pR.bid[103])
  forelem (j; j ∈ pS.sid[R[i].sid])
    ℛ = ℛ ∪ (S[j].sid, S[j].sname, S[j].rating, S[j].age,
                R[i].sid, R[i].bid, R[i].day)
while (row ∈ ℛ)
  print(row.sname);
```

As a first transformation, we apply Loop Collapse to the loop nest. This transformation will also rewrite all references to columns in the program:

```
forelem (i; i ∈ pR×S.(bid^R,sid^S)[(103,sid^R)])
  ℛ = ℛ ∪ (R×S[i].sid^S, R×S[i].sname^S, R×S[i].rating^S,
             R×S[i].age^S, R×S[i].sid^R, R×S[i].bid^R, R×S[i].day^R)
while (row ∈ ℛ)
  print(row.sname^S);
```

Secondly, we perform a Loop Merge transformation to merge the *while* loop consuming the tuples with the *forelem* loop which produces these.

```
forelem (i; i ∈ pR×S.(bid^R,sid^S)[(103,sid^R)])
{
  row = (R×S[i].sid^S, R×S[i].sname^S, R×S[i].rating^S, R×S[i].age^S,
         R×S[i].sid^R, R×S[i].bid^R, R×S[i].day^R)
  print(row.sname^S);
}
```

We observe that of all columns added to each result tuple, only the *sname* column is used. Using the explicit table reduction operator, we reduce the size of the generated result set.

```
forelem (i; i ∈ pR×S.(bid^R,sid^S)[(103,sid^R)])
{
  row = (R×S[i].sname^S)
  print(row.sname^S);
}
```

As a last step, the assignment to *row* could be eliminating by having the *print* statement immediately access the database table.

These transformations drastically reduce the amount of overhead in the database program, that is usually imposed by the API provided by DBMS interfacing libraries. As has been shown in Chapter 2 this overhead can be considerable and removal leads to a further reduction in non-essential instructions executed by a database program. In some cases, this is another factor of 2 reduction on top of the instruction reduction achieved by releasing the dependency on a stand-alone DBMS.

3.4 Example Application of the Transformations

In this section we study an example application of the *forelem* loop and transformations described in this chapter to a code fragment from the RUBiS [75] benchmark. We have performed many of these transformations on the RUBiS benchmark in order to estimate the amount of non-essential instructions performed by web applications shown in Chapter 2. We found that approximately 90% of the executed instructions were non-essential and could thus be eliminated. Elimination of these non-essential instructions had an immediate impact on performance. We showed that because of these reductions in instruction count and execution time, the elimination of non-essential instructions is expected to significantly reduce energy use of server systems running web applications by 70% to 90%.

The following code fragment, written in pseudocode similar to PHP and edited for clarity, is based on the file *ViewUserInfo.php* from the RUBiS benchmark [75]:

```
$commentsResult =
   mysql_query("SELECT * FROM comments WHERE " .
               "comments.to_user_id=$userId");
if (mysql_num_rows($commentsResult) == 0)
  print("<h2>There is no comment for this user.</h3><br>\n");
else
{
  print("<DL>\n");
  while ($commentsRow = mysql_fetch_array($commentsResult))
  {
    $authorId = $commentsRow["from_user_id"];
    $authorResult =
        mysql_query("SELECT nickname FROM users " .
                    "WHERE users.id=$authorId");

    $authorRow = mysql_fetch_array($authorResult);
    $authorName = $authorRow["nickname"];

    $date = $commentsRow["date"];
    $comment = $commentsRow["comment"];
    print("<DT><b><BIG>".
        "<a href=\"/PHP/ViewUserInfo.php?userId=".$authorId.
```

```
        "\>"$authorName</a></BIG></b> wrote the ".$date.
        "<DD><i>".$comment."</i><p>\n");
  }
  print("</DL>\n");
}
```

When the SQL queries that are performed by calling the DBMS API are replaced with *forelem* loop nests which execute in this process, we obtain:

```
forelem (i; i ∈ pComments.to_user_id[$userId])
  ℛ1 = ℛ1 ∪ (comments[i].id, comments[i].from_user_id,
                comments[i].to_user_id, comments[i].item_id,
                comments[i].rating, comments[i].date,
                comments[i].comment)
if (is_empty(ℛ1))
  print("<h2>There is no comment for this user.</h3><br>\n");
else
{
  print("<DL>\n");
  while ($commentsRow ∈ ℛ1)
  {
    $authorId = $commentsRow["from_user_id"];

    forelem (j; j ∈ pUsers.id[$authorId])
        ℛ2 = ℛ2 ∪ (users[j].nickname)

    $authorRow = r2 ∈ ℛ2;
    $authorName = $authorRow["nickname"];

    $date = $commentsRow["date"];
    $comment = $commentsRow["comment"];
    print("<DT><b><BIG>".
        "<a href=\"/PHP/ViewUserInfo.php?userId=".$authorId.
        "\>"$authorName</a></BIG></b> wrote the ".$date.
        "<DD><i>".$comment."</i><p>\n");
  }
  print("</DL>\n");
}
```

As a first transformation, we will merge the *forelem* loop producing the tuples into result set $\mathscr{R}_1$ with the while loop consuming tuples from that result set. To do this, we first have to perform a preparatory transformation, similar to the one outlined in Section 3.3.4. We will move the *if*-statement checking `is_empty` to after the merged loop and change it to check how many result tuples were processed. This is safe, because the true clause of the *if*-statement will only be run if the query loop did not produce (and in the merged case, process) any result tuple. Secondly, the `is_empty` function in the *if*-statement is under our control and we can ascertain that the function does not introduce any side effects.

For the second *forelem* loop, producing into $\mathscr{R}_2$, we observe that consistently only the first result of the set is used. This will be caught by def-use analysis, similar to the example described in Section 3.3.5. We will apply a similar transformation here and use an additional mask column to ensure only one table row is processed by the inner loop. Also in this case, we move the code consuming the result tuples into the inner *forelem* loop body. Code accessing `Comments[i]` is moved as well, which is valid considering `i` is invariant under the inner loop. The increment of `results` can be moved because we know the inner loop will output at least one and at most one tuple.

At the same time we perform a first explicit table reduction and replace the references into result tuples with direct references into the database table, see Section 3.3.10. Applying these transformations results in the following code:

```
$results = 0;
forelem (i; i ∈ pComments.to_user_id[$userId])
{
  forelem (j; j ∈ pUsers.(id,idMask)[(Comments[i].from_user_id, 1)])
  {
    if ($results == 0)
      print("<DL>\n");
    $results++;

    $authorName = Users[j]["nickname"];

    $authorId = Comments[i]["from_user_id"];
    $date = Comments[i]["date"];
    $comment = Comments[i]["comment"];
    print("<DT><b><BIG>".
        "<a href=\"/PHP/ViewUserInfo.php?userId=".$authorId.
        "\>"$authorName</a></BIG></b> wrote the ".$date.
        "<DD><i>".$comment."</i><p>\n");
  }
}
if ($results == 0)
  print("<h2>There is no comment for this user.</h3><br>\n");
else
  print("</DL>\n");
```

For the remainder of the discussion, we will focus solely on the *forelem* loops with the other code removed:

```
$results = 0;
forelem (i; i ∈ pComments.to_user_id[$userId])
{
  forelem (j; j ∈ pUsers.(id,idMask[(Comments[i].from_user_id, 1)])
  {
    $results++;
  }
}
```

In this particular case, it is interesting to perform the Loop Collapse transformation to merge the `Comments` and `Users` tables. For the current discussion, we are not concerned with the cost involved to generate this cross product. As indicated in Section 3.3.9, it is likely that the full table will not be generated due to the implicit table reduction operators that will be applied by the framework. After the merge, we can eliminate the inner *forelem* loop over `Users` and satisfy the query with a single pass over a single table.

We observe the loop is perfectly nested and the Loop Collapse transformation can be applied. The collapsed loop nest looks as follows:

```
$results = 0;
forelem (i; i ∈ pComments×Users.
                          (to_user_id^Comments,id(^Users,idMask^Users)
                          [($userId,from_user_id^Comments,1)])
{
  $results++;
}
```

During the Loop Collapse process, all references to fields of the two tables being merged are rewritten to be references to fields of the combined table. After rewriting the references, the code becomes:

```
$results = 0;
forelem (i; i ∈ pComments×Users.
                          (to_user_id^Comments,id(^Users,idMask^Users)
                          [($userId,from_user_id^Comments,1)])
{
  if ($results == 0)
    print("<DL>\n");
  $results++;

  $authorName = Comments×Users[i].["nickname^Users"];

  $authorId = Comments×Users[i].["from_user_id^Comments"];
  $date = Comments×Users[i].["date^Comments"];
  $comment = Comments×Users[i].["comment^Comments"];
  print("<DT><b><BIG>".
      "<a href=\"/PHP/ViewUserInfo.php?userId=".$authorId.
      "\>"$authorName</a></BIG></b> wrote the ".$date.
      "<DD><i>".$comment."</i><p>\n");
}
if ($results == 0)
  print("<h2>There is no comment for this user.</h3><br>\n");
else
  print("</DL>\n");
```

Through the course of this example, we first eliminated all calls to MySQL's interfacing API and replaced these with equivalents performing the requested operation in place. The calls which request MySQL to evaluate a query, in particular

`mysql_query`, have been replaced with a *forelem* loop nest computing the query. Next, we applied Loop Merge, to merge the producer and consumer loops of the executed queries. This typically saves a single full iteration of the result set. Explicit table reduction was applied to eliminate reads of columns from the tables which were not used by the code consuming the retrieved data. Finally, a Loop Collapse was performed, such that the requests for data are now served from a single table instead of two separate tables.

The end result is translated to C code and compiled into a final executable using a high-end optimizing compiler. By applying this methodology to the file *ViewUserInfo.php*, we have been able to eliminate at least 95% of the instructions executed by the PHP script to generate this web page.

3.5 Conclusions

In this chapter, we presented a methodology to remove the division of application and DBMS codes, so that applications can be optimized to their fullest potential using optimizing compiler technology. The methodology is centered around the *forelem* loop nest, which uses index sets to encapsulate iteration and to simplify the query loop nest so that aggressive compiler optimizations can be successfully applied. Because database queries are expressed in terms of an iterative language using *forelem* loop nests, it allows for the integration of the code performing evaluation of the database query with the application code. The *forelem* constructs have been designed to integrate in a normal optimizing compiler work flow, such that the existing body of optimizations can be re-used.

It is not our intention to move *forelem* forward as a new programming paradigm for programming database applications. Rather, we want our methodology to coexist with existing development environments and frameworks for database application programming. *forelem* is solely used as an intermediate representation in a code optimization backend that is able to take an existing database or web application and automatically breaks down the layers.

We are fully aware that the material presented in this chapter is just a starting point and lots of future work remains. For a full implementation of the integration of DBMS and application codes using *forelem*, several implementation issues need to be addressed. These include the storage layer, sorting, dynamic index set generation, etc. We believe that with this methodology it will become possible to eliminate the majority of the software overhead we described in Chapter 2. Given the huge potential to reduce e.g. energy usage, we think it is definitely worth to explore this area.

CHAPTER 4

Forelem Extensions for Nested Queries

In this chapter we propose a way to express nested queries where subqueries are written in separate functions as *forelem* loop nests. Subsequently, we discuss a number of transformations which are especially suited for *forelem* loop nests representing nested queries. The first of these transformations is to allow a nested query to be rewritten into a single *forelem* loop nest. In other words, the separate functions for subqueries are inlined whenever possible. Another transformation allows for elimination of the use of temporary tables. This puts the loop nest into a form which enables further optimization using transformations as listed above. As a final transformation, we recognize canonical forms of loop nests evaluating nested queries and transformations between these forms. A transition to another canonical form has the potential to enable a whole new dimension of transformations on a loop nest.

4.1 Expressing Subqueries as Procedures

The simplest approach to express nested queries in *forelem* representation of the query is to write these as functions. A query containing a doubly nested query will be translated to *forelem* code which calls a function to perform the first subquery, which subsequently calls another function to perform the second (doubly nested) subquery.

We define a nested query to be a function containing a *forelem* loop nest which will execute the query at the corresponding nesting level. Any references to data or columns in the containing query must be passed as arguments to this function. Functions take zero or more arguments, which are written as $\$0, \$1, ..., \$n$ in the function. The result of evaluating the nested query is returned as a return value. The SQL-92 standard [45] defines three flavors of nested queries: scalar subquery, row subquery and table subquery. This gives rise to three possibilities for return values of functions: a scalar value, a tuple or a multiset.

For example, the query

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid = 103)
```

contains the nested query

```
SELECT R.sid FROM Reserves R WHERE R.bid = 103
```

which is written in *forelem* as:

```
function subquery()
{
  𝒯 = ∅
  forelem (i; i ∈ pR.bid[103])
    𝒯 = 𝒯 ∪ (R[i].sid)
  return 𝒯
}
```

where $\mathscr{T}$ is a temporary table. It is clear that this concerns a table subquery. For scalar subqueries and row subqueries, the return value is simply replaced with a scalar or a tuple respectively.

To see how functions with arguments are handled, let us consider the following query:

```
SELECT R.date
FROM Reserves R
WHERE R.bid = 103 AND EXISTS (SELECT S.sname
                              FROM Sailors S
                              WHERE S.sid = R.sid
                              AND S.sname = "Horatio");
```

where the value `R.sid` is passed to the subquery. The procedure performing the nested query is written as:

```
function subquery($0)
{
  𝒯 = ∅
  forelem (i; i ∈ pS.(sid,sname)[($0,"Horatio")])
    𝒯 = 𝒯 ∪ (S[i].sname)
  return 𝒯
}
```

The main query will iterate the *Reserves* table with an induction variable i and pass the value of `R[i].sid` as the first argument to the procedure `subquery`.

4.2 Expressing Nested Query Operators in forelem

At least as important is the problem of how to express the different operators that can be used in conjunction with the results of nested queries expressed in *forelem* loops. Such operators include `IN`, `EXISTS`, `ANY`, etc. Let us consider the `IN` operator used in our example:

```
S.sid IN (SELECT R.sid
          FROM Reserves R
          WHERE R.bid = 103)
```

This predicate evaluates to true if `S.sid` is in the set resulting from the evaluation of the nested query. Bearing in mind that we can generate an index set for each table in a *forelem* loop nest, including temporary tables, we can write the entire query using *forelem* constructs as follows:

```
function subquery()
{
  𝒯 = ∅
  forelem (i; i ∈ pR.bid[103])
    𝒯 = 𝒯 ∪ (R[i].sid)
  return 𝒯
}

forelem (i; i ∈ pS)
{
  𝒯 = subquery()
  forelem (j; j ∈ is_not_empty(p𝒯.sid[S[i].sid]))
    ℛ = ℛ ∪ (S[i].sname)
}
```

In this code fragment, `is_not_empty` is an index set modifier. The purpose of an index set modifier is to modify the index set without touching the original, so that the iteration space of the *forelem* loop is changed. For example, the `single` modifier modifies the iteration space such that only the first index from the index set is returned. The main advantage of modifiers is that they can be embedded in a *forelem* statement, such that the exact iteration sequence is still encapsulated and *forelem* statements can participate in the various *forelem* loop nest transformations which have been defined. Also, modifiers can be reduced by introducing a mask column in the table. We have discussed mask columns earlier in Section 3.3.5.

The `is_not_empty` modifier will return the first index of the index set when the index set is non-empty, otherwise nothing is returned which will refrain the *forelem* loop from executing its loop body. In fact, the modifier is equal in operation to the `single` modifier, which also returns the first index, but for clarity we maintain both.

The `is_empty` modifier, with the inverse operation, also exists. In this case, however, a dummy index is returned instead of the first index into the index set (since the index set is empty). The dummy index cannot be used to access a table.

For this particular code fragment, the `is_not_empty` modifier will cause the loop body of the *forelem* statement to either be executed once or not be executed at all. Only a single tuple is added to the result set or none.

`NOT IN` can be implemented by simply replacing the `is_not_empty` modifier with `is_empty`. The keyword `EXISTS` tests whether a relation is not empty and can be implemented similar to the example. In this case, the index set $p\mathscr{T}$ will not have a condition attached.

Another class of nested query operators are the `ANY` and `ALL` operators. These operators are combined with an inequality operator such as < or >. A construction such as

```
S.rating > ALL (SELECT S2.sid
                FROM Sailors S2
                WHERE S2.sname = "Horatio")
```

evaluates to true if `S.rating` is larger than all ratings of sailors named "Horatio". Or, put differently, `S.rating` is larger than the largest rating of a sailor named "Horatio". From this follows that it is possible to express nested queries using `ANY` and `ALL` as nested queries which compute a `MAX` or `MIN` aggregate and vice versa, as is described in [36].

Let us consider the query using > `ALL` in full:

```
SELECT S.sid FROM Sailors S
WHERE S.rating > ALL (SELECT S2.rating
                      FROM Sailors S2
                      WHERE S2.sname = "Horatio")
```

The query is to return the *sid* of all sailors who have a rating larger than all sailors named "Horatio". We will employ the is_empty modifier again to express this query using *forelem* loops:

```
function subquery()
{
  𝒯 = ∅
  forelem (i; i ∈ pS.name["Horatio"])
    𝒯 = 𝒯 ∪ (S[i].rating)
  return 𝒯
}

forelem (i; i ∈ pS)
{
  𝒯 = subquery()
  forelem (j; j ∈ is_empty(p𝒯.rating([S[i].rating, ∞))
    ℛ = ℛ ∪ (S[i].sid)
}
```

Please note that the notation is intermixed with `[ , )` denoting closed/open intervals. We have seen the notation `rating[18]` earlier, which denotes that the value of the column *rating* must equal 18. The interval notation specifies that the value

of *rating* must be in the specified interval. In this case, the interval $[\texttt{S[i].rating}, \infty)$ denotes that rating should be equal or larger than `S[i].rating`.

The condition in the query that `S[i].rating` must be greater than the ratings of all sailors named "Horatio" (or rather greater than the highest rating of a sailor named "Horatio") is translated to: there exists no rating for a sailor named "Horatio" which is larger or equal to `S[i].rating`. We test whether the index set on the temporary relation returned by the subquery is empty if we select on ratings equal to or larger than `S[i].rating`. If this index set is empty, we will output a single tuple by performing a single iteration of the inner loop body.

Similarly, `< ALL` translates to checking whether a value is smaller than all values returned by a nested query (or smaller than the minimum value, e.g. the use of a `MIN` aggregate).

Where `> ALL` compares whether a given value is larger than all values (or the maximum) of a given set, `> ANY` compares whether a given value is larger than any value (or the minimum) of a given set. Essentially, maximum is swapped for minimum. Therefore, it is possible to express `ANY` queries using *forelem*. Let's consider the following example:

```
SELECT S.sid FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
                      FROM Sailors S2
                      WHERE S2.sname = "Horatio")
```

which can be written using *forelem* loops as follows:

```
function subquery()
{
  𝒯 = ∅
  forelem (i; i ∈ pS.name["Horatio"])
    𝒯 = 𝒯 ∪ (S[i].rating)
  return 𝒯
}

forelem (i; i ∈ pS)
{
  𝒯 = subquery()
  forelem (j; j ∈ is_not_empty(p𝒯.rating((−∞, S[i].rating])))
    ℛ = ℛ ∪ (S[i].sid)
}
```

The selection logic was transformed to say that there must be a rating in $\mathscr{T}$ which is smaller than or equal to `S[i].rating`, otherwise `S[i].rating` can never be larger than any rating in $\mathscr{T}$.

Although nested queries with `ANY` and `ALL` can be expressed as *forelem* loops without problems, in the literature it is suggested to use a nested query using a `MIN` or `MAX` aggregate function instead to avoid potential confusion [82]. For the last query discussed, `S[i].rating` must be larger than the minimum rating found for a sailor named "Horatio". This can be written in SQL as follows:

```
SELECT S.sid FROM Sailors S
WHERE S.rating > (SELECT MIN(S2.rating)
                  FROM Sailors S2
                  WHERE S2.sname = "Horatio")
```

Note that this subquery returns a scalar value and thus is a scalar subquery, contrary to the `ANY` and `ALL` queries which use table subqueries. This subquery is similarly expressed as a function which returns a scalar value. For the implementation of aggregate functions using *forelem* we refer to Chapter 5.

4.3 Set Operators

The SQL standard [45] also describes set operators, such as `UNION`, `INTERSECT` and `EXCEPT`. Queries using these set operators could be considered nested queries as well, because the query actually consists out of more than one subquery. While SQL queries retain duplicates by default (unless the `DISTINCT` keyword is used), it is important to understand that the set operators do *not* retain duplicates by default.

Let us consider two temporary tables $\mathscr{T}_1$ and $\mathscr{T}_2$. To implement the `UNION` set operator, we merge the two temporary tables and make sure each tuple only appears once. The query can be expressed in *forelem* loops using the *distinct* syntax[1], that will be introduced in Section 5.2, and the `is_empty` modifier introduced in this chapter:

```
𝒯1 = subquery1()
𝒯2 = subquery2()

forelem (i; i ∈ p𝒯1.distinct)
  ℛ = ℛ ∪ (𝒯1[i])
forelem (i; i ∈ p𝒯2.distinct)
  forelem (j; j ∈ is_empty(p𝒯1[𝒯2[i]]))
    ℛ = ℛ ∪ (𝒯2[i])
```

We start by adding all distinct tuples from $\mathscr{T}_1$ to the result set. After that, we add all distinct tuples from $\mathscr{T}_2$ which do not appear in $\mathscr{T}_1$. It is clear that a query in a subquery can be inlined in this loop nest. The temporary tables can possibly be eliminated using the transformation described in Section 4.4.2

The `EXCEPT` operator can be implemented using the same constructs. The goal is to include all tuples in the result set which appear in $\mathscr{T}_1$ but not in $\mathscr{T}_2$. This results in the following code:

```
𝒯1 = subquery1()
𝒯2 = subquery2()

forelem (i; i ∈ p𝒯1.distinct)
```

[1] An index set can be suffixed with the keyword `.distinct`. This has as result that the index set only consists of unique tuples, any duplicate tuple is excluded from the iteration space.

```
  forelem (j; j ∈ is_empty(p𝒯2[𝒯1[i]]))
    ℛ = ℛ ∪ (𝒯1[i])
```

Note that this is similar to a query using `NOT IN`. The `INTERSECT` operator is the opposite and can be compared to a query using the `IN` operator:

```
𝒯1 = subquery1()
𝒯2 = subquery2()

forelem (i; i ∈ p𝒯1.distinct)
  forelem (j; j ∈ is_not_empty(p𝒯2[𝒯1[i]]))
    ℛ = ℛ ∪ (𝒯1[i])
```

4.4 Transformations on Nested Queries

We described several transformations for *forelem* loop nests in Section 3.3. These transformations include Loop Collapse, Loop Interchange and Table Reduction Operators. Using these transformations, the loop nest can be transformed into a more optimal code or a better table layout can be discovered by applying Loop Collapse and Reverse Loop Collapse.

The more loops a *forelem* loop nest contains, the more effective these transformations are. Therefore, for the nested queries discussed in this chapter it is of interest to inline the nested queries such that a single large loop nest exists. In this section we will present transformations to accomplish this and transformations to introduce and eliminate temporary storage. Whether or not temporary storage should be used is a trade-off between memory usage, cache usage and the time needed for calculating the contents of the temporary storage.

4.4.1 Transforming a Nested Query to a Single forelem Loop Nest

Let us consider a singly-nested query, in which we can identify two subqueries S_1 and S_2. This is done according to the form S_1 `WHERE` *field* `IN` (S_2), but also works for operators other than `IN`. As an example, we will re-use the query:

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid = 103)
```

which was introduced earlier in this chapter, where we also presented this query expressed in *forelem* loops. When we inline the procedure performing the subquery in this code fragment, we obtain:

```
forelem (i; i ∈ pS)
{
  𝒯 = ∅
```

```
  forelem (k; k ∈ pR.bid[103])
    𝒯 = 𝒯 ∪ (R[k].sid)
  forelem (j; j ∈ is_not_empty(p𝒯.sid[S[i].sid]))
    ℛ = ℛ ∪ (S[i].sname)
}
```

It is now clearly visible that the index set `pR.bid[103]` is generated for each iteration in the inner loop. The SQL standard [45] mandates that the subquery is *effectively executed* for each evaluation of the `WHERE` clause of the main query. In this case it is obvious that this particular index set, and the resulting table $\mathscr{T}$ after iteration of this index set, will be the same in each iteration of the outer loop (of course as long as the table *Reserves* is not modified). We therefore argue that if the loop generating $\mathscr{T}$ is only performed once, outside of the outer loop, it is still effectively executed for each evaluation of the `WHERE` clause because the intended result, which is always the same for every loop iteration, is always used for each clause evaluation. So, when an analysis can prove that index sets are loop invariant (the subquery is not corresponding) and are free from side effects, more aggressive optimization is possible by moving loop invariant code segments.

4.4.2 Temporary Table Reduction

It will often be the case that a *forelem* loop is producing tuples into a temporary table, which is only read by a consecutive *forelem* loop. In such cases it might be beneficial to eliminate the usage of the temporary table and to merge the two *forelem* loops. This transformation is different from the Loop Merge transformation described in Section 3.3.4, as that particular transformation targets a *forelem* loop and *while* loop. We distinguish two cases of Temporary Table Reduction.

Let us first consider the simple case, in which we have two *forelem* loops, one producing tuples into a temporary table and the second loop fully iterating over this temporary table:

```
𝒯 = ∅
forelem (k; k ∈ pR.bid[103])
  𝒯 = 𝒯 ∪ (R[k].sid)
forelem (j; j ∈ p𝒯)
  forelem (i; i ∈ pS.sid[𝒯[j].sid])
    ℛ = ℛ ∪ (S[i].sname)
```

In this case it is obvious that we can merge the *forelem* loop with iteration counter `j` with the *forelem* loop with iteration counter `k` that produces the tuples that are consumed. Note that the intention here is to merge two *forelem* loops, contrary to the Loop Merge transformation for *forelem* loops described in Section 3.3, where the loop body of a *while* loop iterating the results is merged into the body of the inner loop of a *forelem* loop nest. The resulting code is:

```
forelem (k; k ∈ pR.bid[103])
  forelem (i; i ∈ pS.sid[R[k].sid])
    ℛ = ℛ ∪ (S[i].sname)
```

and the temporary table has been eliminated. The reverse operation, Temporary Table Introduction follows from this by introducing a temporary table and having a *forelem* loop add tuples to this table containing all fields which are used by the *forelem* loops that will consume the tuples from this temporary table.

We now consider a different example where conditions are present on the index set on the temporary table:

```
𝒯 = ∅
forelem (k; k ∈ pR)
  𝒯 = 𝒯 ∪ (R[k].sid, R[k].bid)
forelem (j; j ∈ p𝒯.bid[103])
  forelem (i; i ∈ pS.sid[𝒯[j].sid])
    ℛ = ℛ ∪ (S[i].sname)
```

In this case, the merge is accomplished by first transforming the condition on the temporary table index set to an *if*-statement:

```
𝒯 = ∅
forelem (k; k ∈ pR)
  𝒯 = 𝒯 ∪ (R[k].sid)
forelem (j; j ∈ p𝒯)
  if (𝒯[j].bid == 103)
    forelem (i; i ∈ pS.sid[𝒯[j].sid])
      ℛ = ℛ ∪ (S[i].sname)
```

Now the temporary table can be eliminated analogously to the first example. The reference $\mathscr{T}$[j].bid in the *if*-statement will be written as R[k].bid. Finally, the *if*-statement can simply be transformed into a condition on the index set on *Reserves*, making the resulting code fragment equal to the result of the transformations on the first example.

4.4.3 Modifier Manipulation

We cannot apply Temporary Table Reduction to the example we introduced in Section 4.4.1, because the loop we would like to merge contains an is_not_empty modifier. When an index set is wrapped in a modifier, it is not immediately possible to simply move the conditions to *if*-statements. To correctly evaluate a modifier such as is_not_empty, it is necessary to know the final size of the index set after the conditions have been applied. In order to enable application of the Temporary Table Reduction transformation in such cases, we introduce two transformations to manipulate index sets with conditions within a modifier.

In the example, we observe that the full iteration of index set pR.bid[103] generates the table $\mathscr{T}$. This table is only used by the index set p$\mathscr{T}$.sid[S[i].sid]. In fact, the first index set is being further narrowed down by selecting a specific *sid*. Instead of narrowing down on this specific *sid* within the modifier, we can transfer this condition into a preceding *forelem* loop as follows:

```
forelem (i; i ∈ pS)
{
```

```
  𝒯 = ∅
  forelem (k; k ∈ pR.bid[103])
    𝒯 = 𝒯 ∪ (R[k].sid)
  𝒯₂ = ∅
  forelem (q; q ∈ p𝒯.sid[S[i].sid])
    𝒯₂ = 𝒯₂ ∪ 𝒯[q]
  forelem (j; j ∈ is_not_empty(p𝒯₂))
    ℛ = ℛ ∪ (S[i].sname)
}
```

The condition on the index set iterated by the variable `q` can now be moved to an *if*-statement within the loop body. After that, the temporary table $\mathscr{T}$ can be eliminated, resulting in:

```
forelem (i; i ∈ pS)
{
  𝒯₂ = ∅
  forelem (k; k ∈ pR.bid[103])
    if (R[k].sid == S[i].sid)
      𝒯₂ = 𝒯₂ ∪ R[k]
  forelem (j; j ∈ is_not_empty(p𝒯₂))
    ℛ = ℛ ∪ (S[i].sname)
}
```

and after moving the *if* condition into the index set:

```
forelem (i; i ∈ pS)
{
  𝒯₂ = ∅
  forelem (k; k ∈ pR.(bid,sid)[(103,S[i].sid)])
    𝒯₂ = 𝒯₂ ∪ R[k]
  forelem (j; j ∈ is_not_empty(p𝒯₂))
    ℛ = ℛ ∪ (S[i].sname)
}
```

When we observe that the index set $p\mathscr{T}_2$ is never iterated and only passed to the `is_not_empty` modifier, we can merge the two remaining inner loops, resulting in the compact form:

```
forelem (i; i ∈ pS)
  forelem (j; j ∈ is_not_empty(pR.(bid,sid)[(103,S[i].sid)]))
    ℛ = ℛ ∪ (S[i].sname)
```

The code fragment will output a single tuple for each `S[i].sid` which has reserved a boat with *bid* 103. This equals the original SQL expression. As a result, we have successfully transformed the original nested query to a single perfectly nested *forelem* loop nest. Note that this loop nest looks similar to how joins are expressed in *forelem*, we will discuss this further in Section 4.4.5.

4.4.4 Canonical Forms for Nested Queries

In the previous sections we discussed how a nested query, with its subqueries expressed as functions, can be transformed to a single loop nest and how further transformations are possible on a single loop nest. We highlighted in Section 4.4.1 that transformations that move the execution of the "subquery loop" are possible when we can prove through dependency analysis that the moved code is invariant to the enclosing loop.

When we consider singly-nested queries of which the subquery is loop invariant such that it can be subjected to various transformations, we can define four canonical forms for a singly-nested query with a main query S_1 and subquery S_2. These four forms are shown in Figure 4.1. The starting point, where the subquery is inlined in the body of the loop of the main query, is form 1.

1) **forelem** S_1
   forelem S_2

2) **forelem** S_1
   |
   forelem S_2

3) **forelem** S_2
   forelem S_1

4) **forelem** S_2
   |
   forelem S_1

Figure 4.1: Four canonical forms for nested queries expressed as *forelem* loop nest expressed as *forelem* loop nests.

Loop nest forms 1) and 3) can be transformed into one another using the Loop Interchange transformation for *forelem* loop nests defined Section 3.3.2. The forms 2) and 4) store the results of S_1 or S_2, respectively, in a temporary table. Form 1) can simply be transformed into form 2) by first running the outer loop and storing the results in a temporary table. After that, the inner loop is ran. Basically, this is a transformation which introduces a temporary table in the computation. The reverse, the elimination of a temporary table is also possible. We described this transformation in Section 4.4.2.

By transforming between different canonical forms the search space for the most efficient loop nest is increased. Different canonical forms enable a larger number of possible transformations that can be applied. For example, by moving the subquery loop out of the main loop, such as in form 4), the opportunity is created to possibly merge the subquery loop with any loop preceding it. This can only be done if they operate on the same table. For certain loop nests, however, only moving the subquery loop out of the main loop and storing the results in temporary storage might already be beneficial. Another advantage of the canonical forms is that they allow for loop interchanges; different iteration orders have a different impact on cache utilization.

We have considered the canonical forms for a singly-nested subquery in this section. Using these forms as base case, canonical forms for doubly-nested subqueries and further nesting levels can be considered. Also, forms can be considered for sequences of singly-nested subqueries or subqueries with mixed depth. Naturally, this will result in a larger number of forms, which again implies a larger search space for potentially more efficient loop nests.

Kim [53] identified five basic types of nested queries and described transformations to go from a nested query to a canonical form, which is a canonical n-relation query. The described transformations allow nested queries of arbitrary

depth to be transformed. Ganski et al. present solutions for bugs identified in Kim's transformations, as well as extend the transformations with the ability to handle more nested query operators [36]. The canonical form used by Kim is different from the canonical forms we described in this section. Kim's canonical form is used as a target to transform to, because query processing systems (at that time) can generally process single-level queries performing joins more efficiently than nested queries. Our canonical forms for subqueries are used to extend the search space for efficient loop nests. By transforming a loop nest into a different canonical form, another set of transformations can be applied eventually leading to more efficient code.

4.4.5 Relationship Between `IN` Nested Query Operators and Joins

The possibility to express a nested query using the `IN` operator as a query using a join is well known [53, 36]. However, the original approach described in [53] relies on the assumption that duplicates are to be removed from the results of executing the subquery, in order for the nested query and the query using a join to produce the same final results.

With the queries expressed in *forelem* loop nests, this subtle difference becomes very clear. Recall that at the end of Section 4.4.2 we observed that we transformed a nested query using the `IN` operator into a perfectly nested *forelem* loop, similar to how joins are expressed in *forelem* loop nests. A notable difference is the use of `is_not_empty` in the inner loop. This difference is clearly visible when we compare the loop of Section 4.4.2 with the perfectly nested loop nest representing a join query:

```
1) forelem (i; i ∈ pS)
     forelem (j; j ∈ is_not_empty(pR.(bid,sid)[(103,S[i].sid)]))
       ℛ = ℛ ∪ (S[i].sname)

2) forelem (i; i ∈ pS)
     forelem (j; j ∈ pR.(bid,sid)[(103,S[i].sid)])
       ℛ = ℛ ∪ (S[i].sname)
```

These loops correspond to the following two queries:

```
1) SELECT S.sname                      2) SELECT S.sname
   FROM Sailors S                         FROM Sailors S, Reserves R
   WHERE S.sid IN (SELECT R.sid           WHERE S.sid = R.sid
                   FROM Reserves R        AND R.bid = 103
                   WHERE R.bid = 103)
```

Although both queries answer the same "question", listing all sailor names who have reserved the boat with *bid* 103, the generated result tables are slightly different. Query 1) will output a sailor's name once if its *sid* appears in the list of *sid*s which have reserved boat 103. Query 2) will output a sailor's name for each reservation of boat 103 by that sailor, a sailor's name appears as many times as the sailor reserved boat 103. In other words, in query 1) duplicate reservations

are omitted; query 1) yields a subset of query 2). The duplicate reservations are omitted due to the use of a nested query and the `IN` operator. Note that if two *sid*s exist with the same sailor name who both have reserved boat 103, then this sailor name will still be duplicated!

This difference can be seen in loop nest 1) too. Due to the use of `is_not_empty`, the inner loop is only ran once if a sailor has reserved boat 103; so the sailor name is only output once regardless of the number of reservations. Loop nest 2) can be modified to produce the same result as loop nest 1) easily. The generation of duplicate *sid*s has to be stopped. To constrain this loop to a single iteration, we use the `single` or the equal `is_not_empty` modifier. When these modifiers are used, the loop becomes equal to loop nest 1).

4.5 Example

The following code fragment is based on code taken from Discus. Discus is a web application which has been developed in-house at LIACS and is a complete solution for administration of students, courses, exams and programs. The code fragment is written in pseudocode similar to PHP and edited for clarity. The code is based on a function which generates a listing of courses for which a given student (whose ID is specified through parameter `$id`) is registered. For each course, the main teacher as well as any additional teachers are retrieved.

```
$coursesResult = mysql_query('SELECT * FROM courses, teachers ' .
    'WHERE courses.id IN (select course_id FROM students_courses ' .
    'WHERE student_id = ' . $id . ') AND teacher_id = teachers.id');
while ($row = mysql_fetch_row($coursesResult))
{
  $out = array();
  for ($i = 0; $i < mysql_num_fields($coursesResult); $i++)
  {
    $column = mysql_fetch_field($coursesResult, $i);
    $out[$column->table][$column->name] = $row[$i];
  }
  $courses[] = $out;
}

$i = 0;
foreach ($courses as $course)
{
  $teachersResult =
      mysql_query('SELECT * FROM teachers, teachers_courses ' .
      'WHERE teachers.id = teachers_courses.teacher_id AND ' .
      'teachers_courses.course_id = ' .
      $courses[$i]['courses']['id']);
  while ($row = mysql_fetch_row($teachersResult))
  {
    $out = array();
```

```
    for ($i = 0; $i < mysql_num_fields($teachersResult); $i++)
    {
      $column = mysql_fetch_field($teachersResult, $i);
      $out[$column->table][$column->name] = $row[$i];
    }
    $teachers[] = $out;
  }

  $j = 0;
  foreach ($teachers as $teacher)
  {
    $courses[$i]['teachers'][$j] = $teacher['teachers'];
    $j++;
  }

  $i++;
}
```

As a first step, we replace the usage of the DBMS API with *forelem* loop nests and result sets. We obtain:

```
function subquery($0)
{
  𝒯 = ∅
  forelem (i; i ∈ pStudents_courses.student_id[$0])
    𝒯 = 𝒯 ∪ (students_courses[i].course_id)
  return 𝒯
}


forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    {
      𝒯 = subquery($id)
      forelem (k; k ∈ is_not_empty(p𝒯.course_id[courses[i].id]))
        ℛ1 = ℛ1 ∪ (courses[i].*, teachers[j].*)
    }
while ($row ∈ ℛ1)
{
  $out = array();
  for ($i = 0; $i < len($row); $i++)
  {
    ($table, $column) = get_field_info($row, $i)
    $out[$table][$column] = $row[$i];
  }
  $courses[] = $out;
}
```

```
$i = 0;
foreach ($courses as $course)
{
  $course_id = $courses[$i]['courses']['id'];
  forelem (i; i ∈ pTeachers_courses.course_id[$course_id])
    forelem (j; j ∈ pTeachers.id[teachers_courses[i].teacher_id])
      ℛ2 = ℛ2 ∪ (teachers_courses[i].*, teachers[j].*)
  while ($row ∈ ℛ2)
  {
    $out = array();
    for ($i = 0; $i < len($row); $i++)
    {
      ($table, $column) = get_field_info($row, $i)
      $out[$table][$name] = $row[$i];
    }
    $teachers[] = $out;
  }

  $j = 0;
  foreach ($teachers as $teacher)
  {
    $courses[$i]['teachers'][$j] = $teacher['teachers'];
    $j++;
  }

  $i++;
}
```

In the code fragment a function `get_field_info` is used to get field information. The `$row` variable is a reference to a tuple generated by a *forelem* loop; this tuple does contain information on the tables and columns from which the tuple members originated.

Focusing on the first *forelem* loop nest, we note that the parameter passed to the subquery does not depend on the state of the evaluation of the main query. Due to the absence of such dependencies, the query is non-corresponding and can be evaluated separately. See also the discussion in Section 4.4.1. Therefore, it is possible to inline the function `subquery` into the main loop nest:

```
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    {
      𝒯 = ∅
      forelem (l; l ∈ pStudents_courses.student_id[$id])
        𝒯 = 𝒯 ∪ (students_courses[l].course_id)

      forelem (k; k ∈ is_not_empty(p𝒯.course_id[courses[i].id]))
        ℛ1 = ℛ1 ∪ (courses[i].*, teachers[j].*)
    }
```

The *forelem* loop with iteration counter `l` generates a temporary table which is immediately consumed by the *forelem* loop with iteration counter `k`. This *forelem* loop uses the index set on the temporary table within a modifier. To be able to apply Temporary Table Reduction as described in Section 4.4.2, we first move the conditions on this index set out of the modifier using the transformation described in Section 4.4.3. The first step is to push the condition in the *forelem* loop over $\mathtt{p}\mathscr{T}$ to an *if*-statement:

```
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    {
      𝒯 = ∅
      forelem (l; l ∈ pStudents_courses.student_id[$id])
        𝒯 = 𝒯 ∪ (students_courses[l].course_id)
      𝒯2 = ∅
      forelem (q; q ∈ p𝒯.course_id[courses[i].id])
        𝒯2 = 𝒯2 ∪ 𝒯[q]

      forelem (k; k ∈ is_not_empty(p𝒯2)
        ℛ1 = ℛ1 ∪ (courses[i].*, teachers[j].*)
    }
```

Now that the inner loops are in the form discussed in Section 4.4.2, we apply similar steps to eliminate usage of both $\mathscr{T}_1$ and $\mathscr{T}_2$.

```
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    forelem (k; k ∈ is_not_empty(pStudents_courses.
               (student_id,course_id)[($id, courses[i].id)]))
      ℛ1 = ℛ1 ∪ (courses[i].*, teachers[j].*)
```

If we subsequently merge the *while* loop consuming tuples from $\mathscr{R}_1$ with the obtained *forelem* loop nest, the result is:

```
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    forelem (k; k ∈ is_not_empty(pStudents_courses.
               (student_id,course_id)[($id, courses[i].id)]))
    {
      $row = (courses[i].*, teachers[j].*)
      $out = array();
      for ($i = 0; $i < len($row); $i++)
      {
        ($table, $column) = get_field_info($row, $i)
        $out[$table][$column] = $row[$i];
      }
      $courses[] = $out;
    }
```

We now turn our attention to the second query:

```
forelem (i; i ∈ pTeachers_courses.course_id[$course_id])
  forelem (j; j ∈ pTeachers.id[teachers_courses[i].teacher_id])
    ℛ2 = ℛ2 ∪ (teachers_courses[i].*, teachers[j].*)
```

On this query we will apply the Loop Collapse transformation as described in Section 3.3.5. Important is that the generated cross product is intermediate and will not be generated in full during the code generation phase if it is determined that generating the cross product will be more expensive than an evaluation of the query using two separate tables. More transformations might be done, for example to remove unused columns and rows from the cross product using the Horizontal Iteration Space Reduction and Vertical Iteration Space Reduction transformations described in Sections 3.3.7 and 3.3.8 respectively. Furthermore, the use of `get_field_info` on the collapsed table will continue to refer to the original tables, not the collapsed table, for compatibility.

Next to the Loop Collapse transformation, we also merge the *while* loop consuming the results from $\mathscr{R}_2$ into the loop body where these results are generated.

```
forelem (i; i ∈
      pTeachers_courses×Teachers.(course_id^Teachers_Courses,id^Teachers)
            [($course_id,teacher_id^Teachers_Courses)])
{
  $row = Teachers_courses×Teachers[i];
  $out = array();
  for ($i = 0; $i < len($row); $i++)
  {
    ($table, $column) = get_field_info($row, $i)
    $out[$table][$name] = $row[$i];
  }
  $teachers[] = $out;
}
```

Finally, we will look at the example as a whole again. We will merge the two loops following each *forelem* loop, that is one loop iterating over the `$courses` array generated by the first *forelem* loop and another loop iterating over the `$teachers` array generated by the second *forelem*. We obtain a single loop nest:

```
$i = 0;
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    forelem (k; k ∈ is_not_empty(pStudents_courses.
                  (student_id,course_id)[($id, courses[i].id)]))
    {
      $row = (courses[i].*, teachers[j].*);
      $out = array();
      for ($i = 0; $i < len($row); $i++)
      {
```

```
    ($table, $column) = get_field_info($row, $i)
    $out[$table][$column] = $row[$i];
  }
  $courses[] = $out;

  $j = 0;
  $course_id = $courses[$i]['courses']['id'];
  forelem (ii; ii ∈ pTeachers_courses×Teachers.
                (course_id^Teachers_Courses,id^Teachers)
                [($course_id,teacher_id^Teachers_Courses)])
  {
    $row = Teachers_courses×Teachers[ii];
    $out = array();
    for ($ii = 0; $ii < len($row); $ii++)
    {
      ($table, $column) = get_field_info($row, $ii)
      $out[$table][$name] = $row[$ii];
    }
    $teachers[] = $out;

    $courses[$i]['teachers'][$j] = $teacher['teachers'];
    $j++;
  }

  $i++;
}
```

Further transformations are certainly possible on this loop nest. The calls to `get_field_info` in the final code fragment are really the remains of the use of a SQL API in the original code. Because we have knowledge about the table and column names within the code, we can simply unroll the loops performing these calls. This will result in direct assignments of table data to the output rows, for example:

```
$out['courses']['id'] = $row[0];
```

After this loop unroll, it becomes easier to trace back the assignment statement corresponding to uses of the `$courses` array using def-use analysis. Def-use analysis can also detect that the subscripts of the `$courses` array are appended by adding `$out` variables. The use of the `$out` variable can be eliminated by immediately appending it to the `$courses` array, a transformation which further simplifies the code.

This also paves the way for elimination of assignments from the table data to the `$courses` array for array items, which are never accessed. For example, for the inner *forelem* loops over *Teachers* and *Teachers_Courses*, we will be able to detect that the *Teachers_Courses* fields are never accessed. This information can be used to eliminate the unused *Teachers_Courses* fields from the cross product.

The simplified code after def-use analysis also has the potential to involve the inner two *forelem* loops with the outer three *forelem* loops. Further application of

the Loop Collapse or Loop Interchange transformations could be possible, eventually leading to further data reformatting.

4.6 Conclusions

We have expanded the syntax and set of transformations of *forelem* loops introduced in the previous chapter, with syntax and transformations to handle nested queries. These transformations simplify the analysis of nested queries. So established techniques such as dependency analysis can be used to determine whether or not a subquery has to be executed for every evaluation of the `WHERE` clause, or whether a single execution of the subquery is sufficient to be able to comply with the SQL-92.

By means of an example, we have demonstrated that many potential optimizations exist, that can take advantage of the described transformations. In the example, we were able to merge two separate queries and two separate loops processing the query results into one small and concise loop that is semantically equivalent. Still, there are possibilities to further optimize this loop nest.

CHAPTER 5

Forelem Extensions for Aggregate Queries

This chapter proposes a method to express aggregation queries as *forelem* loop nests. An aggregation query is characterized by function calls into different stages of the aggregate function. These stages are defined, such that they can be used from an *forelem* loop nest to implement an aggregation query. Subsequently, before we can discuss group-by queries which depend heavily on aggregation, we introduce a syntax for working with the *distinct* keyword found in SQL. Typically, duplicate elimination is performed as the last operation during query evaluation. We propose that under several conditions, the *distinct* operation might be moved into the index sets eliminating the separate loop for duplicate elimination. Finally, we introduce a strategy for expressing group-by queries. We show that there are many opportunities to apply the transformations proposed in Chapters 3 and 4. In some cases it is possible to reduce the group-by query to a single *forelem* loop nest.

5.1 Expressing Aggregate Functions

An aggregate function typically has three stages: initialization, update and finalization. The stages serve to initialize any variables, update the variables for each tuple that is processed and to come to a final result. Not all aggregate functions have to implement all three stages. For example, to implement the `COUNT` aggregate, it is sufficient to implement initialization (to set the accumulator variable to zero) and update. To implement `AVG` it is also necessary to implement the finalization stage to perform the division of the sum.

For use within *forelem* loop nests we supply the following functions which represent the stages of an aggregate function:

- `agg_init (handle, agg_func)` initializes the given handle with the given aggregation function.

- `agg_step (handle, agg_func, value)` performs the step stage on the given handle, with the provided aggregate function and value derived from the current tuple.
- `agg_finish (handle, agg_func)` finishes the aggregate computation.
- `agg_result (handle)` returns the computed aggregate value for the handle.

At a later stage in optimization, these functions are replaced with inline variants of the called aggregate function. For the `COUNT` aggregate this means that `agg_init` is replaced with an assignment of the value zero to a variable to initialize the computation, `agg_step` is replaced with a simple value increment and `agg_finish` is replaced with a no-op. By inlining the actual operations, the *forelem* loop nests can be further optimized.

Let us consider the query

```
SELECT AVG (S.age)
FROM Sailors S
```

which performs the *average* aggregate function. We write this query as a *forelem* loop nest as follows using the 4 functions representing the aggregate stages:

```
agg_init(agg1, avg);
forelem (i; i ∈ pS)
  agg_step(agg1, avg, S[i].age);
agg_finish(agg1, avg);
ℛ = ℛ ∪ (agg_result(agg1))
```

When we append a `WHERE` clause to this query, for example `WHERE S.rating = 10`, it is sufficient to replace the use of the index set `pS` in the *forelem* loop with `pS.rating[10]`. Inlining the code performing the different stages of the aggregate function, we obtain:

```
agg1.sum = 0;
agg1.count = 0;
forelem (i; i ∈ pS)
{
  agg1.sum += S[i].age;
  agg1.count++;
}
agg1.result = agg1.sum / agg1.count;
ℛ = ℛ ∪ (agg1.result)
```

From this code sample it is clear that the loop body computes the sum of the vector consisting out of all `age` fields in `S` (the entire table `S` is iterated by index set `pS`). Similarly, the length of this vector is determined by incrementing the `count` variable in the loop body. We described in Section 1.1 that vectorizing compilers will recognize this pattern as reduction operator. The loop thus presents a vectorization opportunity for the optimizing compiler after the *forelem* code has been translated to C code. Without inlining, this opportunity would not have appeared.

5.2 Specification of distinct

Before we can describe how group-by queries are expressed as a *forelem* loop nest, we have to introduce syntax for handling `DISTINCT`. When the `DISTINCT` keyword, referred to as a set quantifier, is specified, redundant duplicate rows will be eliminated from the result table [45]. The keyword always operates on full tuples and it is not possible to perform distinct on a single specified column.

Given a temporary table $\mathscr{T}$, p$\mathscr{T}$.`distinct` specifies the index set on $\mathscr{T}$ that contains unique rows. Duplicates are not present in this index set. Corresponding to the SQL standard [45], the duplicate elimination is performed on the result table. Let us consider the query:

```
SELECT DISTINCT S.sname, S.age
FROM Sailors S
```

This results in:

```
forelem (i; i ∈ pS)
  𝒯 = 𝒯 ∪ (S[i].sname, S[i].age)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ 𝒯[i]
```

In certain cases, it is possible to eliminate the loop iterating over the unique rows of the result set. For this particular example, the loop over `S` does not have any conditions on `pS`. This makes it possible to perform the *distinct* operation when iterating over `pS`. Important is that this operation is applied on just the *sname* and *age* fields which are subsequently projected into the result table, instead of on the full tuples. If the operation is performed on the full tuples, tuples with equal *sname* and *age* but different values for the other fields of the table will still be duplicated in the result table.

The *distinct* syntax assumes by default that the *distinct* operation should be applied to all fields of the table. To limit the operation of the *distinct* keyword to specific fields, one can suffix a tuple of field names to the specification of *distinct* in the index set.

For our example this means that we suffix the *distinct* keyword with the fields *sname* and *age*. This results in the following condensed representation of the same query:

```
forelem (i; i ∈ pS.distinct(sname,age))
  ℛ = ℛ ∪ (S[i].sname, S[i].age)
```

We observe that by applying this transformation, the second loop has been eliminated. Naturally, the reverse transformation is also possible. By moving *distinct* back into a separate loop, application of other transformations is enabled that would otherwise be prevented due to the presence of *distinct* in the index set.

Note that elimination of the second loop for performing duplicate elimination is not always advantageous. Duplicate elimination is an expensive operation that is preferably applied on a table which is as small as possible. In certain cases, moving *distinct* into an index set is beneficial, specially when the operation is moved

to be applied on a smaller table, or when *distinct* is contained in a pre-computed index set.

The correctness of this transformation can be verified using relational algebra[1]. The original loop nest, with an additional loop for eliminating the duplicates, is expressed as:

$$\delta(\pi_{sname,age}(S))$$

In terms of relational algebra, we will express the *distinct* keyword suffixed with specific fields as a projection operation on these specific fields followed by duplicate elimination. The transformed loop nest is then expressed as:

$$\delta(\pi_{sname,age}(S))$$

which equals the expression for the original loop nest.

If in a single-level *forelem* loop all conditions are contained in the index set, it is possible to move the *distinct* operation to the index set. The *distinct* operation must then be limited to fields that will be added, or projected, to the result table. It is clear that this is an extension of the transformation on a loop nest without conditions.

As an example, consider:

```
forelem (i; i ∈ pS.age[18])
  𝒯 = 𝒯 ∪ (S[i].sname)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ (𝒯[i].sname)
```

with the following corresponding relational algebra expression:

$$\pi_{sname}(\delta(\pi_{sname}(\sigma_{age=18}(S))))$$

which can be simplified to:

$$\delta(\pi_{sname}(\sigma_{age=18}(S)))$$

The loop nest can be transformed into the following loop nest:

```
forelem (i; i ∈ pS.distinct(sname).age[18])
  𝒯 = 𝒯 ∪ (S[i].sname)
```

Note that in this syntax the *distinct* operation is performed after the selection, so the loop performs the following expression:

$$\pi_{sname}(\delta(\pi_{sname}(\sigma_{age=18}(S))))$$

where we can eliminate the outer projection again:

$$\delta(\pi_{sname}(\sigma_{age=18}(S)))$$

As a result, this loop is equal to the initial loop nest consisting of two loops.

[1]We use the extended relational algebra proposed by Dayal et. al. [29] which defines relations and operations on these relations in terms of multisets instead of sets. Furthermore, an explicit operator is introduced for elimination duplicates: δ.

If the loop body to which the *distinct* operation is moved contains an *if*-statement, the conditions under which this transformation can be carried out are limited. This makes sense, because the *if*-statement is now performed after the duplicate elimination has been done, contrary to the above example. The *if* statement must resemble a selection operation and when the fields used in the selection do not end up in the result table, the selection test must use the equality operator. The *distinct* operation must be applied to the fields that are added to the result tuple **and** the fields used in the comparison.

To illustrate this, consider the following example corresponding to $\delta(\pi_{sname}(\sigma_{age=18}(S)))$:

```
forelem (i; i ∈ pS)
  if (S[i].age == 18)
    𝒯 = 𝒯 ∪ (S[i].sname)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ (𝒯[i].sname)
```

Transformed into:

```
forelem (i; i ∈ pS.distinct(sname,age))
  if (S[i].age == 18)
    ℛ = ℛ ∪ (S[i].sname)
```

Consider as intermediate step a loop nest which has *distinct* as part of the index set and a separate loop for performing duplicate elimination. The relational algebra expression for such a loop nest is:

$$\delta(\pi_{sname}(\sigma_{age=18}(\delta(\pi_{sname,age}(S)))))$$

This equation can be obtained from the equation corresponding to the original loop by applying the properties of the algebra described in [29]. δ moves past π and δ commutes with σ. Secondly, we are free to remove columns that will not be projected or selected on further on.

To delete the *distinct* operator at the end of the chain (so at the left of the expression), either the projection at the end of the chain does not eliminate any new columns, which is essentially the case handled earlier in this section, or the projection does not introduce any new duplicates. The use of the equality operator during the selection in this case is crucial. The only way two tuples consisting of an *sname* and *age* field with the same values for *sname* can be distinct is to have different values for *age*. Since all tuples will have the value 18 for *age* after selection, all values for *sname* are distinct and the *age* column can be dropped without problems.

Clearly, this does not hold for other operators. Consider the use of the $<$ operator instead of equality. For a selection on $age < 18$ all tuples with $age < 18$ qualify, even if *sname* is equal. After this selection, tuples are present with equal *sname*.

After dropping the *age* column, the result is the following expression, which is indeed equal to the expression corresponding to the transformed loop nest:

$$\pi_{sname}(\sigma_{age=18}(\delta(\pi_{sname,age}(S))))$$

To double-level loop nests similar transformations can be applied Let us consider the following loop nest:

```
forelem (i; i ∈ pB.color["red"])
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯 = 𝒯 ∪ (R[j].bid)
forelem (j; j ∈ p𝒯.distinct)
  ℛ = ℛ ∪ (𝒯[i].bid)
```

With the following corresponding relational algebra expression:

$$\pi_{R.bid}(\delta(\pi_{R.bid}(R \bowtie_{B.bid=R.bid} (\sigma_{B.color="red"}(B)))))$$

We apply properties from [29] to move δ past π and to distribute δ over $\bowtie$:

$$\delta(\pi_{R.bid}(\delta(R) \bowtie_{B.bid=R.bid} \delta(\sigma_{B.color="red"}(B))))$$

To be able to remove the distinct elimination at the end of the chain, we must ensure that the result of the join contains distinct values of *bid* because the final projection is only on *bid*. This is possible when both R and $\sigma_{B.color}(B)$ contain distinct values of *bid* before the join. Because no other fields are needed for the execution of this loop nest, we move the projection inside the distinct eliminations that take place before the join:

$$\pi_{R.bid}(\delta(\pi_{R.bid}(R)) \bowtie_{B.bid=R.bid} \delta(\pi_{B.bid}(\sigma_{B.color="red"}(B)))))$$

This corresponds with the following loop nest:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
    ℛ = ℛ ∪ (R[j].bid)
```

To describe a case where *distinct* cannot be moved to the index set, we consider the query:

```
SELECT DISTINCT R.date
FROM Reserves R
WHERE R.bid = B.bid AND B.color = "red"
```

written in *forelem* as:

```
forelem (i; i ∈ pB.color["red"]))
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯 = 𝒯 ∪ (R[j].date)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ 𝒯[i]
```

Let us look at the corresponding relational algebra expression, where the δ operator has already been distributed over the join:

$$\delta(\pi_{R.date}(\delta(R) \bowtie_{B.bid=R.bid} \delta(\sigma_{B.color="red"}(B))))$$

In order to eliminate δ at the end of the chain, we must project on *B.bid* and *R.bid*,*R.date* before the join. Only the date is projected into the result relation. This final operation will introduce duplicates: consider reservations for a different boat (*bid*) at the same date. So, in this case, we cannot eliminate the second loop performing duplicate elimination.

We can, however, eliminate the separate duplicate elimination loop after first performing a different transformation. When the Loop Collapse transformation described in Section 3.3.5 is performed, the result is:

```
forelem (i; i ∈ pB×R.(color^B,bid^R).[("red", bid^B)]))
  𝒯 = 𝒯 ∪ (B×R[i].date^R)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ 𝒯[i]
```

Now the separate loop for *distinct* can be eliminated by moving the operation to the index set, because we can apply all conditions prior to the duplicate elimination. This also works if the conditions are specified in an *if*-statement instead and we apply *distinct* on all fields used.

5.3 Group-by queries

A group-by query groups tuples of a table by one or more fields, referred to as grouping columns. The values of other columns in the tuples can be aggregated using aggregate functions. Different methods for performing a group-by exist and an appropriate one is usually selected by the query optimizer depending on how table data is to be processed. These methods include performing the grouping operation by hashing and sorting an intermediate table followed by discovering and aggregating the groups.

We do not want to tie ourselves to a particular evaluation strategy for group-by queries, so the exact iteration patterns remain encapsulated in the *forelem* loops. Therefore our aim is to write a group-by query solely using *forelem* loops. In essence, a group-by query iterates over all groups identified by the grouping columns. Three stages are distinguished:

1. A temporary table $\mathscr{T}$ is created containing the selected columns of tuples adhering to an optionally specified `WHERE` clause.
2. The groups are extracted from this temporary table based on the specified grouping columns and stored in $\mathscr{G}$.
3. For each group in turn, we iterate over the group's members stored in $\mathscr{T}$ and perform the requested aggregate functions.

The three stages are written as three *forelem* loop nests. Subsequently, transformations can be applied, such as those described in Chapters 3 and 4. A potential result is that all three loops are merged into a single loop nest.

As an example, let us consider the query:

```
SELECT S.rating, MIN(S.age)
FROM Sailors S
GROUP BY S.rating
```

which we first express as three *forelem* loops:

```
forelem (i; i ∈ pS)
  𝒯 = 𝒯 ∪ (S[i].rating, S[i].age)

forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].rating)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ p𝒯.rating[𝒢[i].rating])
    agg_step(agg1, min, 𝒯[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

We then apply a number of transformations on these loops to attempt to merge them into a single loop nest. In particular we will apply Temporary Table Reduction as described in Section 4.4.2. First, the first loop is duplicated such that the second and third loop nests, each using the results generated by the first loop, get a copy:

```
forelem (i; i ∈ pS)
  𝒯 = 𝒯 ∪ (S[i].rating, S[i].age)
forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].rating)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS)
    𝒯3 = 𝒯3 ∪ (S[j].rating, S[j].age)
  forelem (j; j ∈ p𝒯3.rating[𝒢[i].rating])
    agg_step(agg1, min, 𝒯3[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

Now, we can apply Temporary Table Reduction to eliminate the generation of $\mathscr{T}$ and $\mathscr{T}_3$. For the third loop nest, this is accomplished by moving the index set conditions to *if*-statements, performing the reduction and moving the *if*-statements back to index set conditions.

```
forelem (i; i ∈ pS)
  𝒯2 = 𝒯2 ∪ (S[i].rating)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS.rating[𝒢[i].rating])
    agg_step(agg1, min, S[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

On the first loop nest, it is now possible to eliminate the separate duplicate elimination loop using techniques described in Section 5.2.

```
forelem (i; i ∈ pS.distinct(rating))
  𝒢 = 𝒢 ∪ (S[i].rating)

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS.rating[𝒢[i].rating])
    agg_step(agg1, min, S[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

Finally, another Temporary Table Reduction can be performed to merge both loop nests into one, eliminating the generation of temporary table $\mathscr{G}$.

```
forelem (i; i ∈ pS.distinct(rating))
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS.rating[S[i].rating])
    agg_step(agg1, min, S[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (S[i].rating, agg_result(agg1))
}
```

Next, let us consider a more complicated example involving two tables and a `WHERE` clause:

```
SELECT B.bid, COUNT(*) AS reservationcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = "red"
GROUP BY B.bid
```

As we have discussed, the WHERE clause will be performed by the first loop. We express this query using three *forelem* loops for the three stages as follows:

```
forelem (i; i ∈ pB.color["red"])
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯 = 𝒯 ∪ (B[i].bid, R[j].*)

forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].B.bid)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, count)
  forelem (j; j ∈ p𝒯.B.bid[𝒢[i].B.bid])
    agg_step(agg1, count)
  agg_finish(agg1, count)
  ℛ = ℛ ∪ (𝒢[i].B.bid, agg_result(agg1))
}
```

Note that two fields named *bid* are added to $\mathscr{T}$, to avoid confusion the fields are named *B.bid* and *R.bid*.

We use the same approach as used with the previous example: first duplicate the loop nest generating $\mathscr{T}$, and second use Temporary Table Reduction to merge this in the two remaining loop nests.

```
forelem (i; i ∈ pB.color["red"])
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯2 = 𝒯2 ∪ (B[i].bid)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, count)
  forelem (ii; ii ∈ pB.(bid,color)[(𝒢[i].bid,"red")])
    forelem (jj; jj ∈ pR.bid[B[ii].bid])
      agg_step(agg1, count)
  agg_finish(agg1, count)
  ℛ = ℛ ∪ (𝒢[i].bid, agg_result(agg1))
}
```

Using the technique discussed in Section 5.2 we can eliminate the separate loop performing distinct elimination that follows the first loop nest:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
    𝒢 = 𝒢 ∪ (B[i].bid)

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, count)
  forelem (ii; ii ∈ pB.(bid,color)[(𝒢[i].bid,"red")])
    forelem (jj; jj ∈ pR.bid[B[ii].bid])
      agg_step(agg1, count)
  agg_finish(agg1, count)
  ℛ = ℛ ∪ (𝒢[i].bid, agg_result(agg1))
}
```

Finally, we can eliminate the temporary table $\mathscr{G}$:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
  {
    agg_init(agg1, count)
    forelem (ii; ii ∈ pB.(bid,color)[(B[i].bid,"red")])
      forelem (jj; jj ∈ pR.bid[B[ii].bid])
        agg_step(agg1, count)
    agg_finish(agg1, count)
    ℛ = ℛ ∪ (B[i].bid, agg_result(agg1))
  }
```

5.4 Having keyword

With the *having* keyword a condition can be specified that will be tested against each group. The condition usually only references grouping columns. This condition can only be tested after all members of a group have been processed. The condition cannot be moved into the index set of the enclosing loop.

As an example, we can extend the query used in the previous section to include a `HAVING` clause, specifying that only boats with more than 5 reservations should appear in the result table:

```
SELECT B.bid, COUNT(*) AS reservationcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = "red"
GROUP BY B.bid
HAVING COUNT(*) > 5
```

Because a `COUNT` aggregate is already performed, we do not have to introduce an additional aggregate computation. Before the tuple is added to the result set, we add a test for the *having* condition:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
  {
    agg_init(agg1, count)
    forelem (ii; ii ∈ pB.(bid,color)[(B[i].bid,"red")])
      forelem (jj; jj ∈ pR.bid[B[ii].bid])
        agg_step(agg1, count)
    agg_finish(agg1, count)
    if (agg_result(agg1) > 5)
      ℛ = ℛ ∪ (B[i].bid, agg_result(agg1))
  }
```

After this addition, the *forelem* loop nest now computes the desired result.

5.5 Example

In this section we demonstrate how the techniques discussed in this chapter can be applied to a real-world code example. The following code fragment is based on the file *AboutMe.php* from the RUBiS [75] benchmark. The code fragment is written in pseudocode similar to PHP and edited for clarity.

```
$bidsResult =
    mysql_query("SELECT item_id, bids.max_bid FROM bids, items
                 WHERE bids.user_id=$userId AND bids.item_id=items.id
                 AND items.end_date > NOW()
                 GROUP BY item_id");
if (mysql_num_rows($bidsResult) == 0)
    print("<h2>You did not bid on any item.</h2>\n");
else
{
    print("<h3>Items you have bid on.</h3>\n");

    while ($bidsRow = mysql_fetch_array($bidsResult))
    {
        $maxBid = $bidsRow["max_bid"];
        $itemId = $bidsRow["item_id"];
        $itemResult =
            mysql_query("SELECT * FROM items WHERE id=$itemId");

        $currentPriceResult =
            mysql_query("SELECT MAX(bid) AS bid FROM bids ".
                        "WHERE item_id=$itemId");
        $currentPriceRow = mysql_fetch_array($currentPriceResult);
        $currentPrice = $currentPriceRow["bid"];
        if ($currentPrice == null)
            $currentPrice = "none";
```

```
        $itemRow = mysql_fetch_array($itemResult);

        $itemName = $itemRow["name"];
        $itemInitialPrice = $itemRow["initial_price"];
        $quantity = $itemRow["quantity"];
        $itemReservePrice = $itemRow["reserve_price"];
        $startDate = $itemRow["start_date"];
        $endDate = $itemRow["end_date"];
        $sellerId = $itemRow["seller"];

        $sellerResult =
            mysql_query("SELECT nickname FROM users " .
                        "WHERE id=$sellerId")

        $sellerRow = mysql_fetch_array($sellerResult);
        $sellerNickname = $sellerRow["nickname"];

        print("<TR><TD>" .
              "<a href=\"/PHP/ViewItem.php?itemId="
              .$itemId."\">".$itemName.
              "<TD>".$itemInitialPrice."<TD>".$currentPrice."<TD>"
              .$maxBid."<TD>".$quantity.
              "<TD>".$startDate."<TD>".$endDate.
              "<TD><a href=\"/PHP/ViewUserInfo.php?" .
              "userId=".$sellerId."\">".$sellerNickname.
              "</a>\n");

        mysql_free_result($sellerResult);
        mysql_free_result($currentPriceResult);
        mysql_free_result($itemResult);
    }
    mysql_free_result($bidsResult);
}
```

As a first step, all SQL queries that are performed by calling the DBMS API are replaced with *forelem* loop nests which execute in process. The code fragment contains four queries. We will rewrite these queries as *forelem* loop nests and perform preliminary transformations on these queries in turn. After that, we place the loop nests into the code fragment. The first query is:

```
SELECT item_id, bids.max_bid FROM bids, items
WHERE bids.user_id=$userId AND bids.item_id=items.id
AND items.end_date >= NOW()
GROUP BY item_id
```

This query is written as a *forelem* loop nest using the strategy discussed in Section 5.3:

```
forelem (i; i ∈ pBids.user_id[$userId])
  forelem (j; j ∈ pItems.(id,end_date)[(Bids[i].item_id,[NOW(),∞)])
    𝒯 = 𝒯 ∪ (Bids[i].item_id, Bids[i].max_bid)

forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].item_id)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  forelem (j; j ∈ p𝒯.item_id[𝒢[i].item_id])
    r = (𝒯[j].item_id, 𝒯[j].max_bid)
  ℛ = ℛ ∪ r
}
```

Note that $[NOW(), \infty)$ indicates the range in which the value of field *end_date* must lie. And with the described transformations, we can write the query as a single loop nest:

```
forelem (i; i ∈ pBids.distinct(item_id).user_id[$userId])
  forelem (j; j ∈ pItems.distinct(id).(id,end_date)
                              [(Bids[i].item_id,[NOW(),∞))])
  {
    forelem (ii; ii ∈ pBids.(user_id,item_id)
                                [($userId,Bids[i].item_id)])
      forelem (jj; jj ∈ pItems.(id,end_date)
                                  [(Bids[ii].item_id,[NOW(),∞))])
        r = (Bids[ii].item_id, Bids[ii].max_bid)
    ℛ = ℛ ∪ r
  }
```

The second query to be considered is:

```
SELECT * FROM items WHERE id=$itemId
```

which is written as:

```
forelem (i; i ∈ pItems.id[$itemId])
  ℛ = ℛ ∪ Items[i]
```

The third query contains an aggregate function:

```
SELECT MAX(bid) AS bid FROM bids WHERE item_id=$itemId
```

Using the technique described in Section 5.1 we can express the query using *forelem* loops as follows:

```
agg_init(agg1, max);
forelem (i; i ∈ pBids.item_id[$item_id])
  agg_step(agg1, max, Bids[i].bid);
agg_finish(agg1, max);
ℛ = ℛ ∪ (agg_result(agg1))
```

The aggregate operation can subsequently be inlined:

```
agg1.result = 0;
forelem (i; i ∈ pBids.item_id[$item_id])
  if (agg1.result == 0 || agg1.result < Bids[i].bid)
    agg1.result = Bids[i].bid;
ℛ = ℛ ∪ (agg1.result)
```

Finally, the fourth query:

```
SELECT nickname FROM users WHERE id=$sellerId
```

is easily converted to:

```
forelem (i; i ∈ pUsers.id[$sellerId])
  ℛ = ℛ ∪ (Users[i].nickname)
```

We now rewrite the code fragment with the *forelem* loops for the four queries:

```
forelem (i; i ∈ pBids.distinct(item_id).user_id[$userId])
  forelem (j; j ∈ pItems.distinct(id).(id,end_date)
                        [(Bids[i].item_id,[NOW(),∞))])
  {
    forelem (ii; ii ∈ pBids.(user_id,item_id)
                          [($userId,Bids[i].item_id)])
      forelem (jj; jj ∈ pItems.(id,end_date)
                            [(Bids[ii].item_id,[NOW(),∞))])
        r = (Bids[ii].item_id, Bids[ii].max_bid)
    ℛ1 = ℛ1 ∪ r
  }
if (is_empty (ℛ1))
    print("<h2>You did not bid on any item.</h2>\n");
else
{
    print("<h3>Items you have bid on.</h3>\n");

    while ($bidsRow ∈ ℛ1)
    {
        $maxBid = $bidsRow["max_bid"];
        $itemId = $bidsRow["item_id"];

        forelem (i; i ∈ pItems.id[$itemId])
          ℛ2 = ℛ2 ∪ Items[i];
```

```
        agg1.result = 0;
        forelem (i; i ∈ pBids.item_id[$item_id])
          if (agg1.result == 0 || agg1.result < Bids[i].bid)
            agg1.result = Bids[i].bid;
        ℛ3 = ℛ3 ∪ (agg1.result);

        $currentPriceRow = r ∈ ℛ3;
        $currentPrice = $currentPriceRow["bid"];
        if ($currentPrice == null)
            $currentPrice = "none";

        $itemRow = r ∈ ℛ2;

        $itemName = $itemRow["name"];
        $itemInitialPrice = $itemRow["initial_price"];
        $quantity = $itemRow["quantity"];
        $itemReservePrice = $itemRow["reserve_price"];
        $startDate = $itemRow["start_date"];
        $endDate = $itemRow["end_date"];
        $sellerId = $itemRow["seller"];

        forelem (i; i ∈ pUsers.id[$sellerId])
          ℛ4 = ℛ4 ∪ (Users[i].nickname)

        $sellerRow = r ∈ ℛ4;
        $sellerNickname = $sellerRow["nickname"];

        print("<TR><TD>" .
              "<a href=\"/PHP/ViewItem.php?itemId="
              .$itemId."\">".$itemName.
              "<TD>".$itemInitialPrice."<TD>".$currentPrice."<TD>"
              .$maxBid."<TD>".$quantity.
              "<TD>".$startDate."<TD>".$endDate.
              "<TD><a href=\"/PHP/ViewUserInfo.php?" .
              "userId=".$sellerId."\">".$sellerNickname.
              "</a>\n");
    }
}
```

We now apply Loop Merge to merge the *forelem* loop producing the tuples into result set $\mathscr{R}_1$ with the while loop consuming these tuples. Before this transformation can be applied, we must perform a preparatory transformation that moves the *if*-statement checking `is_empty` after the merged loop. The statements in the *else* clause before the while loop are moved into the loop body and made conditional. At the same time we perform an explicit table reduction which replaces references into the result set with direct references into the database table. Subsequently, Global Forward Substitution can be performed. This reduction is also

applied on the result set $\mathscr{R}_3$.

```
results = 0;
forelem (i; i ∈ pBids.distinct(item_id).user_id[$userId])
  forelem (j; j ∈ pItems.distinct(id).(id,end_date)
                          [(Bids[i].item_id,[NOW(),∞))])
  {
    forelem (ii; ii ∈ pBids.(user_id,item_id)
                              [($userId,Bids[i].item_id)])
      forelem (jj; jj ∈ pItems.(id,end_date)
                               [(Bids[ii].item_id,[NOW(),∞))])
        r = (Bids[ii].item_id, Bids[ii].max_bid)

    if (results == 0)
      print("<h3>Items you have bid on.</h3>\n");

    results++;

    forelem (iii; iii ∈ pItems.id[Bids[ii]["item_id"]])
      𝓡2 = 𝓡2 ∪ Items[iii]

    agg1.result = 0;
    forelem (iii; iii ∈ pBids.item_id[Bids[ii]["item_id"]])
      if (agg1.result == 0 || agg1.result < Bids[iii].bid)
        agg1.result = Bids[iii].bid;
    $currentPrice = agg1.result;

    if ($currentPrice == null)
        $currentPrice = "none";

    $itemRow = r ∈ 𝓡2;

    $itemName = $itemRow["name"];
    $itemInitialPrice = $itemRow["initial_price"];
    $quantity = $itemRow["quantity"];
    $itemReservePrice = $itemRow["reserve_price"];
    $startDate = $itemRow["start_date"];
    $endDate = $itemRow["end_date"];
    $sellerId = $itemRow["seller"];

    forelem (iii; iii ∈ pUsers.id[$sellerId])
      𝓡4 = 𝓡4 ∪ (Users[iii].nickname)

    $sellerRow = r ∈ 𝓡4;
    $sellerNickname = $sellerRow["nickname"];

    print("<TR><TD>" .
```

```
          "<a href=\"/PHP/ViewItem.php?itemId="
          .$itemId."\">".$itemName.
          "<TD>".$itemInitialPrice."<TD>".$currentPrice."<TD>"
          .$maxBid."<TD>".$quantity.
          "<TD>".$startDate."<TD>".$endDate.
          "<TD><a href=\"/PHP/ViewUserInfo.php?" .
          "userId=".$sellerId."\">".$sellerNickname.
          "</a>\n");
  }
}
if (results == 0)
  print("<h2>You did not bid on any item.</h2>\n");
```

Further optimizations are possible. For example, def-use analysis will detect that only a single row of $\mathscr{R}_2$ is used. The analysis will also detect that the condition `id == Bids[ii]["item_id"]` holds for all tuples iterated by iteration counter `jj`. Therefore, this *forelem* loop is unnecessary and the data can simply be obtained from `Items[jj]` instead.

Also from result set $\mathscr{R}_4$ a single tuple is used. Therefore, the loop generating this result set can be pruned to only iterate once. This can be accomplished either by using the *single* modifier described in Section 4.2 or by using an additional mask column as described in Section 3.3.5. After this transformation, explicit table reduction can be applied on this loop.

Finally, the fact that the tables *Bids* and *Items* are closely used together might indicate that the Loop Collapse transformation, described in Section 3.3.5 can be of use here. This will eliminate the two joins currently present in the loop nest and might open the road to further transformations. As an example, this has the potential to make it possible to eliminate the query computing the `MAX(bid)` aggregate.

5.6 Conclusions

In this chapter we demonstrated how aggregation queries can be written in terms of a *forelem* loop and introduced a strategy for expressing group-by queries as *forelem* loops. A syntax for duplicate elimination was introduced together with conditions under which the duplicate elimination can be moved to the *forelem* loops' index sets. We have demonstrated that the transformations introduced in the preceding chapters can be applied. Whereas a group-by query is first written as three *forelem* loop nests, it is in certain cases possible to transform this to a single *forelem* loop nest.

By means of an example, we have demonstrated that many potential optimizations exist that can take advantage of the described strategies and transformations. In the example, we were able to merge a code fragment containing a group-by query and three other queries into a single loop nest. Subsequently, the possibility was shown how one of the queries can be fully eliminated. There are further possibilities to optimize this loop nest for example by restructuring the tables using Loop Collapse.

CHAPTER 6

Query Optimization Using the Forelem Framework

6.1 Introduction

This chapter explores the optimization of database queries using just simple compiler transformations. This optimization process is carried out by the consecutive application of simple compiler transformations that are expressed in the *forelem* intermediate representation as a series of *forelem* loops. So instead of using a traditional query optimizer which optimizes queries that have been expressed into an initial query execution plan, the compiler transformations are the main query optimization transformations. This approach is different from other compiler-based approaches to query optimization, such as [57, 73], that focus on code generation and propose compiler-based techniques for the generation of efficient executable code from algebraic query execution plans.

The optimization methodology that is proposed in this chapter is part of a larger framework for the vertical integration of database applications. Extensive vertical integration is not possible with traditional query optimization techniques, because when code is generated from query evaluation plans and combined with application code, further applicability of compiler transformations is obscured. Therefore, it is important that queries are expressed, optimized and combined with application code in a way that compiler optimizations can still be successfully exploited. The *forelem* framework provides such a way. Vertical integration will be more thoroughly discussed in Chapter 7.

The techniques described in this chapter build upon the *forelem* intermediate representation and the transformations introduced in the previous chapters. First, a number of transformations will be introduced that are specific to the optimization of queries expressed in the *forelem* intermediate representation. Secondly, the application of the transformations will be illustrated. Thirdly, strategies will be discussed for the sequence in which the transformations should be applied, as well as strategies for the generation of efficient (C/C++) code from the optimized

forelem intermediate representation of the query. Finally, using the TPC-H benchmark [91], it is demonstrated that queries optimized using compiler transformations in the *forelem* framework have a performance that is comparable to that of contemporary database systems that employ traditional query optimization.

6.2 Specific Forelem Transformations for Query Optimization

This section builds upon the *forelem* loop and transformations introduced in the preceding chapters. In particular, in this chapter use will be made of the Loop Invariant Code Motion, Loop Interchange and Loop Fusion transformations, which will not be reiterated here. For more details, we refer the reader to Section 3.3.

A number of different compiler transformations are specifically introduced for the optimization of queries expressed in the *forelem* intermediate representation. Although these transformations support optimization of queries within the *forelem* framework, these transformations can also be applied in general to optimize overall performance. The main contribution of this section is that the majority of the techniques used for query optimization within the *forelem* framework can be derived from existing optimizing compiler transformations.

6.2.1 Inline

The Inlining transformation inlines a function into its caller. This transformation is commonly used to inline calls to short functions and methods to save the overhead of performing a function call, or to enable further optimization by considering the code of the inlined function in the context of the code that calls this function.

In the *forelem* framework, all subqueries are initially expressed as separate functions. After inlining a subquery into its caller, the subquery can be considered together with the surrounding loops in the caller. For example, subqueries are often called from a loop nest and after inlining the compiler might detect that the subquery is invariant to the loop body from which it is called. As a result, the Loop Invariant Code Motion transformation will move the subquery out of the loop.

Consider the following subquery and loop:

```
subquery0()
{
  count = 0;
  forelem (ii; ii ∈ pA.field1[value])
    count++;
  return count;
}

forelem (i; i ∈ pB)
{
  tmp = subquery0()
```

```
  if (B[i].field2 < tmp)
    ℛ = ℛ ∪ (B[i].field1)
}
```

the subquery is inlined into the caller as follows:

```
forelem (i; i ∈ pB)
{
  count = 0;
  forelem (ii; ii ∈ pA.field1[value])
    count++;
  tmp = count;
  if (B[i].field2 < tmp)
    ℛ = ℛ ∪ (B[i].field1)
}
```

This has enabled subsequent transformations to be applied. The loop computing the `count` variable is invariant to the loop iterating the table `B`. The subquery was "uncorrelated". Because of this, the loop computing `count` can be moved out of the loop:

```
count = 0;
forelem (ii; ii ∈ pA.field1[value])
  count++;
forelem (i; i ∈ pB)
{
  tmp = count;
  if (B[i].field2 < tmp)
    ℛ = ℛ ∪ (B[i].field1)
}
```

6.2.2 Iteration Space Expansion

Within the *forelem* framework a transformation known as Iteration Space Expansion is defined. This transformation is inspired by the Scalar Expansion transformation, which is typically used to enable parallelization of loop nests. There is also a relation with the expansion of the iteration spaces to transform irregular access patterns into regular ones [95]. This transformation is briefly described in this subsection, for a more detailed description see Section 12.3.

Iteration Space Expansion expands the iteration space of a *forelem* loop by removing conditions on its index set. For a loop of the form, with `SEQ` denoting a sequence of statements:

```
forelem (i; i ∈ pA.field[X])
  SEQ;
```

the following steps are performed:

1. the condition `A[i].field == X` is removed, which expands the iteration space so that the entire array `A` is visited,

2. scalar expansion is applied on all variables that are written to in the loop body denoted by *SEQ* and references to these variables are subscripted with the value tested in the condition, in this case `A[i].field`,

3. all references to the scalar expanded variables after the loop are rewritten to reference subscript `X` of the scalar expanded variable.

6.2.3 Table Propagation

The Table Propagation transformation is similar to Scalar Propagation that is typically performed by compilers. In Scalar Propagation, the use of variables whose value is known at compile-time is substituted with that value. For example, in:

```
int x = 3;
int y = x + 3;
int z = x * 9;
```

the uses of `x` can be replaced with the value of `x`, `3`:

```
int x = 3;
int y = 3 + 3;
int z = 3 * 9;
```

In Table Propagation, the use of a temporary table of which the contents are known is replaced with a loop nest that generates the same contents as the temporary table. This eliminates unnecessary copying of data to create the temporary table, but also enables further transformations because the loop nest that generates the contents of the temporary table can now be considered together with the loop nest that iterates the temporary table. For example, consider the following *forelem* loops:

```
forelem (i; i ∈ pX.field2[value])
  𝒯 = 𝒯 ∪ (X[i].field1)

forelem (i; i ∈ p𝒯)
  forelem (j; j ∈ pY.field2[𝒯[i].field1])
    ℛ = ℛ ∪ (Y[j].field1)
```

The first loop generates a table $\mathscr{T}$, which is iterated by the second loop. The table $\mathscr{T}$ is being "streamed" between these consecutive loops. Table $\mathscr{T}$ can be propagated to the second loop nest:

```
forelem (i; i ∈ pX.field2[value])
  𝒯 = 𝒯 ∪ (X[i].field1)

forelem (i; i ∈ pX.field2[value])
  forelem (j; j ∈ pY.field2[X[i].field1])
    ℛ = ℛ ∪ (Y[j].field1)
```

The result of the first loop, table $\mathscr{T}$, is now unused. Therefore, the first loop may be eliminated by a succeeding compiler transformation, resulting in:

```
forelem (i; i ∈ pX.field2[value])
  forelem (j; j ∈ pY.field2[X[i].field1])
    ℛ = ℛ ∪ (Y[j].field1)
```

This final result gives the impression that a variant of Loop Fusion has been applied. Rather, in the *forelem* framework this optimization is expressed as two separate transformations: Table Propagation and Dead Code Elimination. Note that this transformation is a generalized form of the Temporary Table Reduction transformation discussed in Section 4.4.2.

6.2.4 Dead Code Elimination

Dead Code Elimination removes statements whose results are not used in any subsequent statements. Such statements can be detected using, for example, def-use analysis. In the *forelem* framework, tables are treated as variables. As a result, statements that generate tables that are unused in the remainder of the *forelem* representation of the problem will be removed by Dead Code Elimination.

6.2.5 Index Extraction

The Index Extraction transformation extracts the use of an index set from a *forelem* statement. A new loop is created that iterates the index set and fills a temporary table. The index set in the original loop is replaced with an unconditional iteration of this temporary table. This transformation will transform the following loop:

```
forelem (i; i ∈ pTable1.field1[value1])
  SEQ;
```

into:

```
forelem (i; i ∈ pTable1.field1[value1])
  𝒯 = 𝒯 ∪ (...)

forelem (i; i ∈ p𝒯)
  SEQ;
```

In fact, this transformation can be seen as the opposite of Table Propagation. The Index Extraction transformation extracts one or more *forelem* loops from a loop nest to a new loop nest that generates a temporary table. The original loop nest is modified to replace the extracted loops with a loop iterating the temporary table.

This transformation is useful in the following example:

```
forelem (i; i ∈ pTable1.(field2,field3)[(value1, value2)])
  forelem (j; j ∈ pTable2.field1[Table1[i].field1])
    forelem (k; k ∈ pTable3.field1[Table2[j].field2])
      forelem (l; l ∈ pTable4.field1[Table3[k].field1])
        if (Table4[l].field2 == value3)
          SEQ;
```

where SEQ denotes a sequence of statements. In this case, the transformations decided to move the tests of the conditions on Table1 to the outermost loop, because two conditions are tested and potentially prunes the search space by a large extent. Due to the dependences between the other tables, the condition for Table4 is only tested in the inner loop.

Suppose that Table4 and subscript l are not used in SEQ, then the iteration of this array is not necessary in this loop nest. Instead, the subscripts k that should be iterated can be computed before executing this loop nest. This is done by executing the inner two loops and finding all subscripts k, for which a subscript l exists that satisfies the condition Table4[l].field2 == value3. The results of this computation are stored in a temporary table, along with other fields from Table3 that are referenced in SEQ. This operation results in:

```
forelem (k; k ∈ pTable3.field1)
  forelem (l; l ∈ pTable4.field1[Table3[k].field1])
    if (Table4[l].field2 == value4)
      𝒯 = 𝒯 ∪ (Table3[k].field1)

forelem (i; i ∈ pTable1.(field2,field3)[(value2, value3)])
  forelem (j; j ∈ pTable2.field1[Table1[i].field1])
    forelem (k; k ∈ p𝒯.field1[Table2[j].field2])
      SEQ;
```

Note that all references in SEQ to Table3 must be rewritten to refer to $\mathscr{T}$ instead.

6.3 Example

This section illustrates the usage of the transformations, by applying these on query 13 from the TPC-H benchmark[1]. The SQL code for query 13 is as follows:

```
select c_count,
  count(*) as custdist
from (
    select c_custkey,
      count(o_orderkey)
    from
      customer left outer join orders on
      c_custkey = o_custkey
      and o_comment not like '%express%requests%'
    group by
      c_custkey
) as c_orders (c_custkey, c_count)
group by
  c_count
order by
```

[1]Query 13 was chosen because of its size and usefulness to serve as an illustration. The other queries would have taken up too much space

```
  custdist desc,
  c_count desc;
```

When this query is translated into the *forelem* intermediate representation, the result is:

```
subquery0()
{
 forelem (i; i ∈ pCustomer) {
   forelem (j; j ∈ pOrders) {
     if (customer[i].c_custkey == orders[j].o_custkey &&
        !like(orders[j].o_comment, "%express%requests%")
       𝒯 = 𝒯 ∪ (customer[i].c_custkey, orders[j].o_orderkey)
      else
       𝒯 = 𝒯 ∪ (customer[i].c_custkey, nil)
   }
 }

 forelem (i; i ∈ p𝒯) {
   𝒢 = 𝒢 ∪ (𝒯[i].c_custkey)
 }
 distinct(𝒢)

 forelem (i; i ∈ p𝒢) {
   count = 0;
   forelem (j; j ∈ p𝒯.c_custkey[𝒢[i].c_custkey]) {
     if (𝒯[j].o_orderkey != nil)
       count++;
   }
   𝒮 = 𝒮 ∪ (𝒢[i].c_custkey, count)
 }

 return 𝒮;
}

𝒮 = subquery0();
forelem (i; i ∈ p𝒮) {
  𝒯₂ = 𝒯₂ ∪ (𝒮[i].c_count)
}
forelem (i; i ∈ p𝒯₂) {
  𝒢₂ = 𝒢₂ ∪ (𝒯₂[i].c_count)
}
distinct(𝒢₂)

forelem (i; i ∈ p𝒢₂) {
  count = 0;
  forelem (j; j ∈ p𝒯₂.c_count[𝒢₂[i].c_count]) {
    count++;
  }

  ℛ = ℛ ∪ (𝒢₂[i].c_count, count)
}
```

As a first step, the Inline transformation is performed, which will inline the subquery at the point where the subquery is called. For the above example this is trivial. Subsequently, the Loop Interchange transformation is considered. The only loop nest where Loop Interchange could possibly be applied is the loop nest over the `Customer` and `Orders` tables. Usually, Loop Interchange would be applied at this location such that the condition on `o_comment` can be tested in the outer loop. Note that in this case, the body of the *if* statement depends on both loop iterators and as such the statement cannot be moved to the other loop.

The next transformation that can be applied on this example is Table Propagation. In the first step, the loop creating table $\mathscr{T}$ is propagated to the consecutive loop nest accessing $\mathscr{T}$ and the loop creating $\mathscr{T}_2$ is propagated to the loops accessing $\mathscr{T}_2$.

```
forelem (i; i ∈ pCustomer) {
  forelem (j; j ∈ pOrders) {
    if (customer[i].c_custkey == orders[j].o_custkey &&
       !like(orders[j].o_comment, "%express%requests%")
      𝒯 = 𝒯 ∪ (customer[i].c_custkey, orders[j].o_orderkey)
     else
      𝒯 = 𝒯 ∪ (customer[i].c_custkey, nil)
  }
}

forelem (i; i ∈ pCustomer) {
  forelem (j; j ∈ pOrders) {
    if (customer[i].c_custkey == orders[j].o_custkey &&
       !like(orders[j].o_comment, "%express%requests%")
      𝒢 = 𝒢 ∪ (customer[i].c_custkey)
     else
      𝒢 = 𝒢 ∪ (customer[i].c_custkey)
  }
}
distinct(𝒢)

forelem (i; i ∈ p𝒢) {
  count = 0;
  forelem (j; j ∈ p𝒯.c_custkey[𝒢[i].c_custkey]) {
    if (𝒯[j].o_orderkey != nil)
      count++;
  }
  𝒮 = 𝒮 ∪ (𝒢[i].c_custkey, count)
}

forelem (i; i ∈ p𝒮) {
  𝒯2 = 𝒯2 ∪ (𝒮[i].c_count)
}
forelem (i; i ∈ p𝒮) {
  𝒢2 = 𝒢2 ∪ (𝒮[i].c_count)
}
distinct(𝒢2)
```

```
forelem (i; i ∈ p𝒢₂) {
  count = 0;
  forelem (j; j ∈ p𝒮.c_count[𝒢₂[i].c_count]) {
    count++;
  }

  ℛ = ℛ ∪ (𝒢₂[i].c_count, count)
}
```

The transformation must be repeated several times for all propagations to be resolved. The result of the repeated application of the transformation is:

```
forelem (i; i ∈ pCustomer) {
  forelem (j; j ∈ pOrders) {
    if (customer[i].c_custkey == orders[j].o_custkey &&
       !like(orders[j].o_comment, "%express%requests%")
      𝒯 = 𝒯 ∪ (customer[i].c_custkey, orders[j].o_orderkey)
     else
      𝒯 = 𝒯 ∪ (customer[i].c_custkey, nil)
  }
}

forelem (i; i ∈ pCustomer) {
  forelem (j; j ∈ pOrders) {
    if (customer[i].c_custkey == orders[j].o_custkey &&
       !like(orders[j].o_comment, "%express%requests%")
      𝒢 = 𝒢 ∪ (customer[i].c_custkey)
     else
      𝒢 = 𝒢 ∪ (customer[i].c_custkey)
  }
}
distinct(𝒢)

forelem (i; i ∈ p𝒢) {
  count = 0;

  forelem (ii; ii ∈ pCustomer.c_custkey[𝒢[i].c_custkey]) {
    forelem (jj; jj ∈ pOrders) {
       if (customer[ii].c_custkey == orders[jj].o_custkey &&
         !like(orders[jj].o_comment,"%express%requests%")
         row = (customer[ii].c_custkey, orders[jj].o_orderkey)
       else
         row = (customer[ii].c_custkey, nil)
       if (row.o_orderkey != nil)
         count++;
    }
  }

  𝒮 = 𝒮 ∪ (𝒢[i].c_custkey, count)
}

forelem (i; i ∈ p𝒮) {
```

```
  𝒯₂ = 𝒯₂ ∪ (𝒮[i].c_count)
}
forelem (i; i ∈ p𝒮) {
  𝒢₂ = 𝒢₂ ∪ (𝒮[i].c_count)
}
distinct(𝒢₂)

forelem (i; i ∈ p𝒮.distinct(c_count)) {
  count = 0;
  forelem (j; j ∈ p𝒮.c_count[𝒮[i].c_count]) {
    count++;
  }

  ℛ = ℛ ∪ (𝒮[i].c_count, count)
}
```

Dead Code Elimination will remove statements that produce statements of which the results are not used:

```
forelem (i; i ∈ pCustomer) {
  𝒢 = 𝒢 ∪ (customer[i].c_custkey)
}
distinct(𝒢)

forelem (i; i ∈ p𝒢) {
  count = 0;

  forelem (ii; ii ∈ pCustomer.c_custkey[𝒢[i].c_custkey]) {
    forelem (jj; jj ∈ pOrders) {
       if (customer[ii].c_custkey == orders[jj].o_custkey &&
         !like(orders[jj].o_comment, "%express%requests%")
         count++;
    }
  }

  𝒮 = 𝒮 ∪ (𝒢[i].c_custkey, count)
}

forelem (i; i ∈ p𝒮.distinct(c_count)) {
  count = 0;
  forelem (j; j ∈ p𝒮.c_count[𝒮[i].c_count]) {
    count++;
  }

  ℛ = ℛ ∪ (𝒮[i].c_count, count)
}
```

Note that in case all transformations are repeated, Table Propagation will propagate the loop iterating `Customer` and generating a table $\mathscr{G}$, to the consecutive loop accessing $\mathscr{G}$.

6.4 Optimization and Code Generation Strategies

In order to successfully optimize *forelem* loop nests using the transformations described in Section 6.2, a strategy is needed that determines in which order the transformations on the *forelem* loop nests are to be performed. The *forelem* framework uses the following strategy:

- First, subqueries are inlined, so that these can be considered in combination with the calling context.
- As a second step loops are reordered such that as many conditions as possible are tested in the outermost loops. Priority is given to move conditions that test against a constant value to the outermost loop. This step is a combination of the application of Loop Interchange with Loop Invariant Code Motion.
- Thirdly, opportunities for the application of Iteration Space Expansion are looked for. An example of such an opportunity is a loop iterating an index set with a condition on a field, of which the body computes an aggregate function. Iteration Space Expansion is followed by Loop Invariant Code Motion, because the loop computing the aggregate function is often made loop invariant by the Iteration Space Expansion transformation. Iteration Space Expansion is not applied on loops iterating temporary tables.
- The fourth step is to apply Table Propagation to prepare for the elimination of unnecessary temporary tables.
- Fifth, Index Extraction is performed on inner loops that iterate tables that could be removed from the loop nest.
- Finally, Dead Code Elimination is performed to remove any loop that computes unused results.

Experiments have been conducted with the queries from the TPC-H benchmark [91]. The different transformations that have been applied to each TPC-H query during the *forelem* optimization phase are shown in Table 6.1.

Another optimization strategy is to perform a brute-force exploration of the entire optimization space. This is useful, for example, for queries that are run many times on changing data so that the costly optimization effort is worth it. We plan to study brute-force exploration of the optimization search space in future work.

Code generation

Next to strategies for the application of transformations on the *forelem* intermediate representation, there are also strategies for the generation of efficient code from the *forelem* intermediate representation. These strategies are for a large part concerned with the selection of *forelem* loops for which index sets should be generated at run-time and the selection of efficient data structures for such index sets. The following rules are used for the code generation of index sets:

Query #	Applied Transformations
1	Table Propagation, LICM, Dead Code Elimination
2	Inline, Loop Interchange, LICM, Iteration Space Expansion, LICM, Index Extraction
3	Loop Interchange, LICM
4	Inline, Index Extraction
5	Loop Interchange, LICM, Index Extraction
6	None
7	Loop Interchange, LICM, Index Extraction
8	Loop Interchange, LICM, Index Extraction
9	Loop Interchange, LICM, Index Extraction, Table Propagation, Dead Code Elimination
10	Loop Interchange, Table Propagation
11	Inline, Loop Interchange, LICM, Table Propagation, Dead Code Elimination
12	Loop Interchange, LICM
13	Inline, Table Propagation, Dead Code Elimination
14	None
15	Inline, Loop Interchange, LICM, Table Propagation, Dead Code Elimination
16	Inline, Loop Interchange, LICM, Table Propagation, Dead Code Elimination
17	Iteration Space Expansion, LICM
18	Loop Interchange, LICM, Table Propagation, Dead Code Elimination
19	None
20	Inline, Iteration Space Expansion, LICM
21	Loop Interchange, LICM
22	Inline, LICM

Table 6.1: An overview of the transformations applied to each TPC-H query, in the order of application. The abbreviation LICM stands for Loop Invariant Code Motion.

1. Index sets without conditions address the fully array. No index set is generated in this case, instead the full array is iterated with subscripts $i \in [0, len)$.
2. Index sets that are used in multiple loop nests get priority in being generated.
3. The index set of the outer loop is never explicitly generated, as the outer loop is only iterated once.
4. For very small tables, index sets are not generated.

Different data structures are used as index set, such as flat arrays, hash tables or tree structures, depending on the properties of the index set. For example, if it is known that the field, for which an index set is created, has a unique value for each row in the array, a one-to-one-mapping is set up using a flat array or hash table. This property can be known to the code generator because the field was specified as primary key in the table schema, or the generated code detects at run-time that the table data satisfies this condition. For index sets that yield multiple subscripts balanced tree is used.

Additionally, the code generator can easily generate both row-wise and column-wise data access code. Within the *forelem* framework, a change from row-wise to column-wise layout is a trivial transformation. Which layout should be used is determined by the amount of fields in an array that are accessed.

6.5 Experimental Results

Experiments have been conducted using the queries from the TPC-H benchmark [91]. All queries were parsed into the *forelem* intermediate representation, optimized using the transformations described in this chapter and C/C++ code has been generated from the optimized AST. These executables access the database data through memory-mapped I/O. The execution time of the queries is compared to the execution time of the same queries as executed by PostgreSQL [80] and MonetDB [69].

All experiments have been carried out on an Intel Core 2 Quad CPU (Q9450) clocked at 2.66 GHz with 4 GB of RAM. The software installation consists out of Ubuntu 10.04.3 LTS (64-bit), which comes with PostgreSQL 8.4.9. The version of MonetDB used is 11.11.11 (Jul2012-SP2), which is the latest version that could be obtained from the MonetDB website [69] for use with this operating system.

On a TPC-H data set of scale factor 1.0, all queries were run with PostgreSQL, *forelem*-generated code and MonetDB. The execution times of the different queries in milliseconds are shown in Figure 6.1. PostgreSQL queries that took longer than 30 seconds to complete have been omitted from the figure for clarity. In the majority of cases, the *forelem*-optimized implementations have an execution time in the same order of magnitude as MonetDB, in a few cases even surpassing it.

MonetDB and the *forelem*-generated code, have also been tested on a dataset with scale factor 10.0. The execution times of the different queries in seconds are shown in Figure 6.2. In more than half of the queries, the *forelem*-optimized code performs the query with performance comparable to or faster than MonetDB.

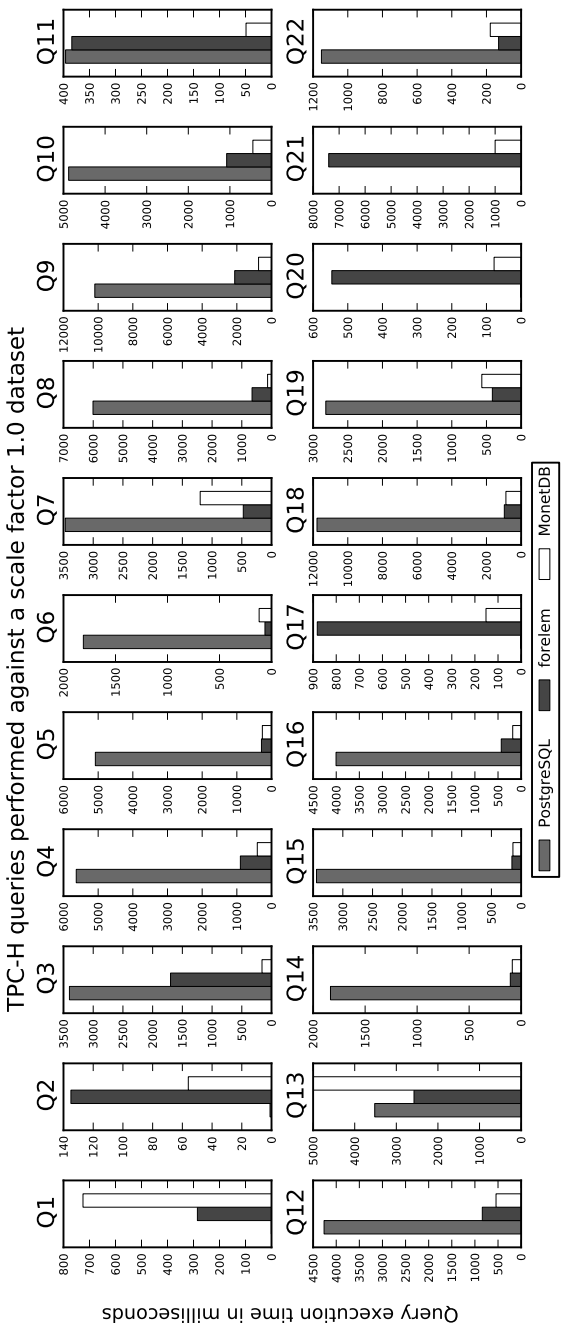


Figure 6.1: Execution time in milliseconds of the TPC-H queries performed against a scale factor 1.0 dataset. The PostgreSQL results for Q1, Q2, Q17, Q20, Q21 were all beyond 31 seconds and were omitted from the figure for clarity.

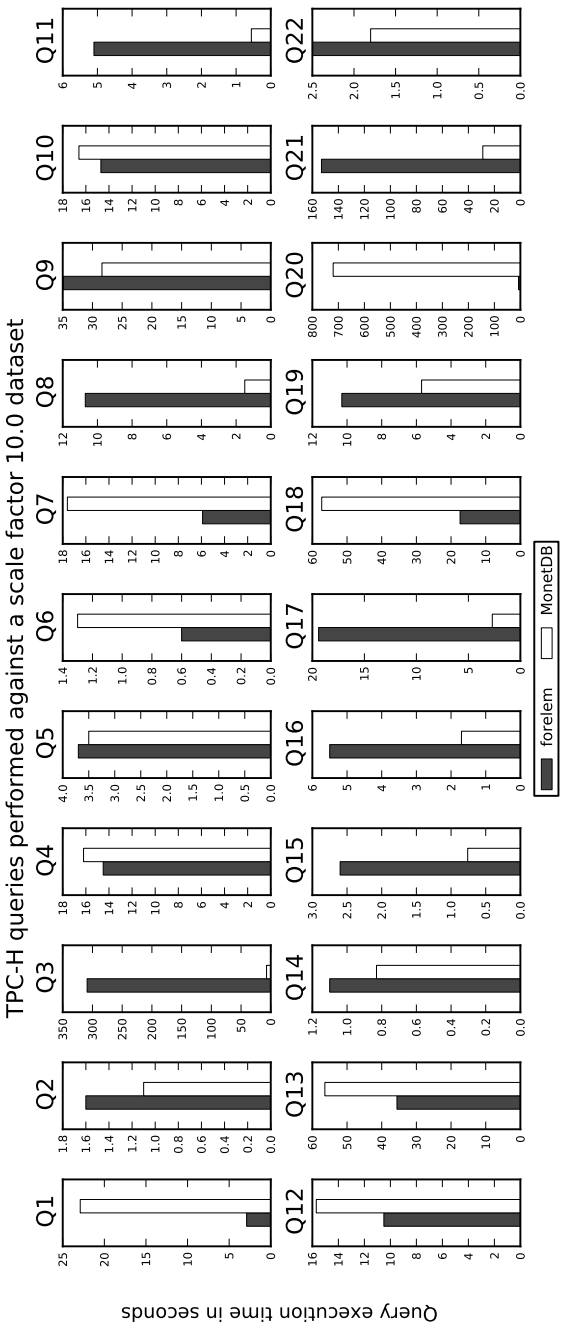


Figure 6.2: Execution time in seconds of the TPC-H queries performed against a scale factor 10.0 dataset.

MonetDB is clearly faster in a third of the queries. We intend to address this gap in future work, by improving the used optimization strategies.

6.6 Conclusions

In this chapter, the optimization of database queries using compiler transformations has been described. This optimization process is carried out in the *forelem* framework. The *forelem* framework provides an intermediate representation in which queries can be naturally expressed and on which compiler transformations can be applied to optimize the loop nest. Compiler transformations that are currently implemented within the *forelem* framework were illustrated and strategies for the application of these transformations were discussed.

Experimentation using the queries from the TPC-H benchmark shows that the queries that were optimized using compiler transformations within the *forelem* framework are capable of achieving similar performance to that of contemporary database systems. However, while the *forelem* framework has been designed to provide full integral optimization, the *forelem* framework is still able to reach performance comparable to contemporary database systems.

CHAPTER 7

Automatically Reducing Database Applications To Their Essence

7.1 Introduction

In this chapter, we propose an optimization technique specifically targeting the minimization of the number of instructions[1]. Essentially, the idea behind the technique is to locate and eliminate unnecessary instructions. These are instructions that can be omitted without affecting the course of execution and the output of an application. As a result of this elimination, an application is reduced to its essence. As can be concluded from Chapter 2, the number of instructions of the resulting executables can be significantly reduced when database applications are reduced to their essence. Note that this approach does not take cycles per instruction (CPI) into account at this point, but solely focuses on the reduction of the number of instructions. If the CPI remains around the same level, this implies a reduction of execution time by the same amount, directly improving the performance of the software. More likely, however, is that the CPI will slightly increase. This is attributed to the fact that the instructions performing memory traffic to and from the database tables, instructions characterized by a higher CPI, are not eliminated. Nevertheless, if the CPI would increase by 50%, then still up to 92.5% of the total number of cycles to be executed is eliminated. This is a drastic improvement and as a consequence the targeted hardware platform is more effectively exploited.

The large number of instructions that can be eliminated from database applications stems from that fact that these applications are typically developed with a modular approach. At the foundation a database management system (DBMS) is used and these systems have traditionally been developed as separate, independent software (server) applications. This independence makes database systems modular and enables their use in a variety of applications. Between database

[1]Note that traditional compiler optimizations which target code compaction should not be confused with the target of this chapter. Code compaction could still be used on the resulting codes from our optimizations.

systems and database applications various framework layers are often used to facilitate development. This layered and modular approach allows for rapid prototyping, development and deployment.

However, this approach does come at a price. It is well known that the cost of the overhead induced by this modular and layered approach is significant. More importantly, the stacking of layers obscures the essence of the database application. The essence of the application can only be captured by breaking down these layers. This has as result that the number of instructions is drastically reduced, having a direct, very advantageous, effect on the application's performance. Although the fact that overhead is created by these layered approaches might be obvious, the amount of overhead that is induced by these methods, up to 95% of the total amount of instructions, is rather surprising (see later on in this chapter).

Another consequence of this stacking is that compiler optimizations are mostly restricted to the application part of database applications while the optimization of the DBMS server is mostly delegated to the query optimizer. In fact, there is generally no integration of these two optimization efforts. If a query optimizer is not aware of how the data is used within the application, or if the application optimizer cannot influence optimization of the data access done by the query optimizer, a database application can never be optimized to its full potential automatically. Therefore, exposing the essence of a database application has as additional advantage that application optimization and query optimization can be targeted integrally, further increasing the performance of the application. In our approach, both the application and its queries are optimized using optimizing compiler techniques [104]. Optimizing compilers have been very effective in high performance computing as well as in general computing by optimizing loop structures, data structures, register allocation, data prefetching, etc.

The process of capturing the essence of the application consists out of automatically stripping the layers of which a database application is built up. These layers include a high-level development (scripting) language, frameworks that facilitate rapid development in that language, the DBMS API layer, and so on. By eliminating these layers, the essence of the application is parsed into a common (compiler) intermediate representation. In this intermediate representation, all database accesses are exposed as accesses to arrays of structures, governed by simple loop control structures. On the other hand, this approach allows current development methodologies for DBMS applications to remain in place. For example, current development environments and frameworks to develop Java-based database applications or PHP-based web applications have been and are serving programmers very well. The reduction process as proposed in this chapter will be part of the backend code development process, so that the DBMS application development methodology will not be directly affected by the reduction process. Rather, an application is developed and tested as usual, but before extensive deployment the code is passed through the code optimization backend to eliminate as much overhead as possible. This way, we continue to take advantage of the available software development tools which enhance programmer productivity and combine this with a code optimization backend that significantly reduces the number of instructions to be executed, thereby also improving the performance of the application.

The effectiveness of the proposed reduction scheme is validated using two web applications: RUBBoS [74] and RUBiS [75]. Both web application benchmarks have been developed by a collaboration between Rice University and INRIA. We show that on average 75% of the instructions can be eliminated, and in specific cases up to 95%, without affecting the execution and output of the application.

The remainder of this chapter is organized as follows. In Section 7.2 the results of the initial study presented in Chapter 2, in which the instructions executed by RUBBoS and RUBiS benchmarks were manually reduced, are briefly reiterated. Section 7.3 describes the methods underlying the automatic instruction reduction process. Section 7.4 discusses how these methods are implemented and can be deployed within an operational workflow. In Section 7.5, we validate the effectiveness of our approach by presenting the results generated with our prototype compiler. Section 7.6 describes further optimizations that are possible on top of the results reported in this chapter. Section 7.8 presents our conclusions.

7.2 Attainable Results

Chapter 2 presented an initial study into the reduction of unnecessary instructions in database applications, among which the RUBBoS and RUBiS benchmarks. As a result, we have shown the potential reductions of instructions that are possible by hand optimizing these applications. For the purpose of identifying which instructions were eliminated, we categorized the non-essential instructions as "PHP overhead", "MySQL overhead" and "SQL API overhead".

Figure 7.1, presents the quantification of the different categories for the RUBBoS and RUBiS benchmark in terms of the number of instructions. As can be seen from this figure, the reduction ranges from 70.2% to 93.3%. From this figure follows that up to 93.3% of instructions can be eliminated, without affecting the results of the program. A direct result of this drastic reduction of the number of instructions is a significant improvement in execution time and energy consumption. For more details, the reader is referred to Chapter 2.

7.3 Vertical Integration

The results described in the previous section serve as a target for an automatic optimization method. This automation is achieved by breaking down the layers from which database applications are commonly built up. We refer to this process as "vertical integration". This section describes how the different layers are broken down automatically.

7.3.1 PHP layer

PHP scripts are typically executed by parsing this script, followed by code generation and execution, resulting in a start-up overhead. Furthermore, PHP code cannot be integrated with Apache and the various PHP database modules, which are written in C. As a first stage in vertical integration, the PHP code is translated

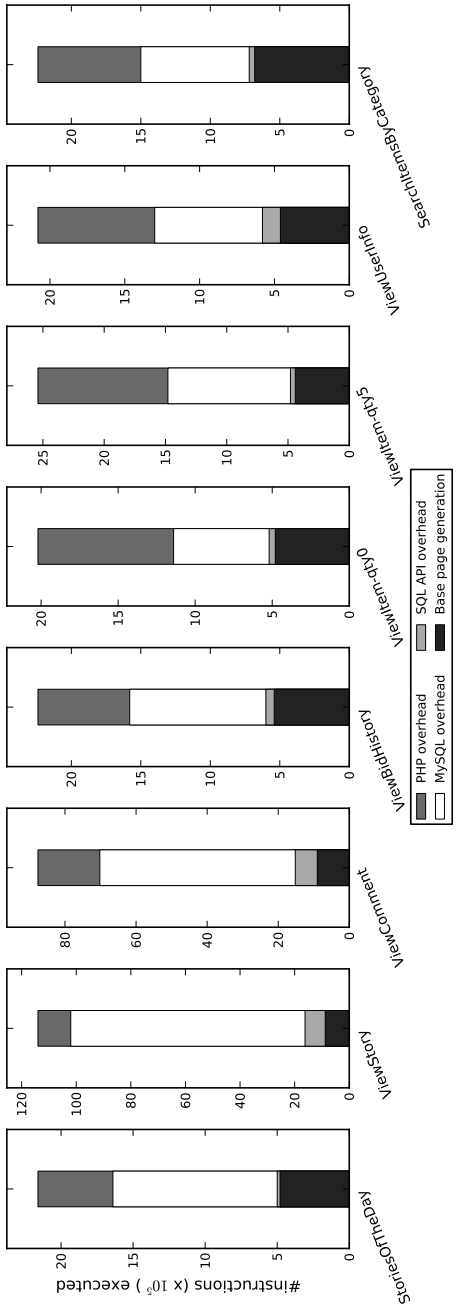


Figure 7.1: Quantification of the amount of executed non-essential instructions, in 10^5 of instructions, to generate a single page for different components from the RUBBoS and RUBiS benchmarks.

to C++ code. This translation thus serves two purposes: parsing and code generation overhead is eliminated and further vertical integration with the database modules is enabled.

To accomplish this code translation, our toolchain contains a source-to-source translator, which translates PHP source code to C++ source code. The source-to-source compiler was extracted from the HipHop for PHP project, that has been developed by Facebook [42]. The result of the automatic invocation of the HipHop source-to-source translator is a generated C++ code, which is subsequently compiled and linked against the HipHop runtime. This runtime contains implementations of the PHP built-in functions and data types that are used by the generated C++ code. It also features a built-in web server, which replaces the typical use of the Apache HTTP server. The web server is vertically integrated into the executable, further reducing the code size.

7.3.2 DBMS layer

The result of the previous step is a C++ source code, that performs calls to a DBMS API to execute SQL queries. This can in fact be compared to an Embedded SQL code. In this step, a vertical integration is performed of the database application and the DBMS that performs data accesses based on the submitted queries. As part of our prototype compiler, we have developed a source-to-source translator that scans the C++ code for calls to a given DBMS API, and replaces the use of this API with C++ code. At this moment, only the MySQL API is supported, but extensions to support other DBMS APIs are straightforward.

This replacement implies that calls to the DBMS API that submit a SQL query for execution are also replaced with code performing the identical operation. To achieve this the *forelem* intermediate representation is used, in which SQL queries can be expressed as a series of simple loop structures. This intermediate representation has been designed such that it integrates well in the workflow of traditional optimizing compilers. After parsing a SQL query into an Abstract Syntax Tree (AST), a *forelem* loop is generated immediately from this AST. No query plan is generated to support this translation. Subsequently, queries are optimized with loop transformations, rather than sophisticated query planning. Another advantage of the design of this intermediate representation is that it allows for straightforward integral optimization of the application code and its queries. Further optimizations that are possible within this framework, but not yet accomplished automatically, are described in Section 7.6.

Our source-to-source translator will detect DBMS API calls in the C++ code. Important calls to detect are for example: opening a connection to the DBMS, sending a query to the DBMS, retrieving the result set, accessing the result set and releasing the result set. The translator will annotate the semantics of these operations in the C++ AST. Commonly, to be able to submit a query to a DBMS, a parameter must be passed that specifies the DBMS connection to use. The values of these parameters are only known at runtime. Therefore, the code translator performs an advanced static analysis on the database application code, in order to deduce which connection is used in which DBMS API call. Similarly, such an analysis is used to find exact query strings that are passed to the DBMS calls.

Query strings that are constructed at runtime in the database applications, are composed out of multiple calls to string manipulation routines.

As soon as all necessary data has been collected, the DBMS API calls can be translated. A prerequisite for a successful translation of a call to execute a query in the DBMS is that the connection is known as well as the query string. The connection information is used to obtain table metadata and the data of the tables at a later stage. Using *libforelem*, a library we developed that can manipulate *forelem* ASTs, the SQL query is parsed. A check is done whether the query conforms to the database schemas. For example, the following SQL query taken from RUBiS:

```
SELECT item_id, bids.max_bid
FROM bids, items
WHERE bids.user_id=$userId
AND bids.item_id=items.id
GROUP BY item_id
```

is written in *forelem* as:

```
forelem (i; i ∈ pBids.user_id[$userId])
  forelem (j; j ∈ pItems.id[Bids[i].item_id])
    𝒯 = 𝒯 ∪ (Bids[i].item_id, Bids[i].max_bid)
forelem (i; i ∈ p𝒯)
  𝒢 = 𝒢 ∪ (𝒯[i].item_id)
forelem (i; i ∈ p𝒢.distinct(item_id))
{
  forelem (j; j ∈ single(p𝒯.item_id[𝒢[i].item_id]))
    ℛ = ℛ ∪ (𝒯[j].item_id, 𝒯[j].max_bid)
}
```

In this loop nest, the notation `pBids.user_id[$userId]` denotes an index set. This index set only contains subscripts `i` into table `Bids` for which the `user_id` value of the tuple equals the value in the variable `$userId`. The *forelem* library will determine how to generate code for this index set. If an index is defined on the table in the original SQL database, an explicit index will be generated in the data generation phase, that is kept updated when writes are done to this table. In case an explicit index is not defined, *libforelem* may choose to insert code before the loop to generate the necessary index set for this loop at runtime.

The *forelem* loop does not have a particular iteration order. The order in which subscripts are stored in the index set, or the order in which these are iterated, is not defined. Because of this, we are not limited in the range of transformations and iteration schemas we can apply. In fact, index sets are the essence of *forelem* loop nests as they encapsulate iteration and simplify the query loop code so that aggressive compiler optimizations can be successfully applied.

A large variety of SQL queries can currently be expressed in terms of the *forelem* intermediate representation. A special syntax is available for expressing the use of aggregate functions. For the *distinct* keyword, a *distinct* tag exists for the index sets. So, application of duplicate elimination will not complicate the expression of the loop in *forelem*. This way, loop transformations can still be applied to the loop effectively. Joins are simply represented by nested *forelem* loops.

Group by queries can be expressed in *forelem* using aggregates, the *distinct* tag and by using multiple *forelem* loops. This results in an initial expression of the query in *forelem*, like the example shown above, which is subsequently subjected to transformations at the *forelem* level.

Observe that the structure of the loop allows for straightforward integration with imperative application codes. Our source-to-source translation extends the C++ AST with the *forelem* AST, so that code transformations can be defined that target both the C++ as well as the *forelem* code. After these transformations have been performed, C++ source code is generated for the *forelem* code when the C++ source file is rewritten. The C++ code that is generated from the *forelem* loop accesses the tables through a simple array of structures. Note, that such transformations are not possible if an existing DBMS were simply integrated into the same process as the application program. This does not result in the DBMS API being broken down. To be able to reduce the maximum amount of instructions, interpretation of the DBMS API calls and the executed queries is a necessity.

The result of this query inlining is typically a loop that generates a result set, see the $\mathscr{T}$, $\mathscr{G}$ and $\mathscr{R}$ sets in the example above. Any suitable data structure can be used to store this result set. The data structure used can be adapted to the characteristics of the query, used tables and the application itself. For example, if only a few results are expected, a more efficient data structure can be used to store these results, contrary to the use of more advanced data structures for storing large result sets. DBMS API calls that retrieve and access the result set are translated into C++ codes that operate on the result set generated by the inlined query code. For a more detailed discussion the reader is referred to Chapter 2.

After all uses of DBMS API have been translated by the source-to-source translator, the translator will determine which database tables and indices are used by the translated queries. Because the operations performed by the DBMS are being integrated into the application program, the accessed data must be migrated as well. The used tables and indices are fetched from the DBMS and stored into local binary files as arrays of structures, that are accessed by the application program using memory-mapped I/O.

7.3.3 DBMS API layer

In the initial generated code, the influence of the DBMS API can still be found. For example, at the original location of a DBMS query call, a loop evaluating the query and generating a result set can now be found. A bit further on in the generated source file, a loop will be found that accesses this result set. These two loops can be merged, eliminating the need to explicitly create a result set. Code transformations like these become possible now that the DBMS API layer has been removed and the application code and queries are expressed in a common intermediate. Currently, our prototype optimizer does not perform such transformations yet.

7.4 Incorporation in an Operational Workflow

Current development methodologies to develop database applications are serving programmers very well. A change in methodology to have the developers focus

on instruction reduction manually is not cost effective. In general, hand optimization of code is an elaborate and expensive task and may not weigh up to the costs that are potentially saved. A clear advantage of our approach is that no modification to current development methodologies is needed and the translation to a significantly more compact code is performed fully automatically. This automatic translation process can be easily integrated into an operational workflow as part of the deployment phase where a new version of the application is to be deployed on the production servers.

Many deployments of web applications are distributed, because a single server can typically not handle the load. However, once a database application has been vertically integrated, it operates on a data store that is local and private to that application. Update actions, i.e. insert, delete and update statements, that are performed by the application are applied on this local data store. For these updates to become visible to other clients of the same database, this data must be distributed to these clients. So, where database systems store data at a centralized location and provide access to this data through an API, vertically integrated applications store data locally and need a method to distribute updates.

A solution is needed to deal with the absence of a (remote) central data store. A straightforward solution is to introduce a central data store in addition to the local data stores and submit all updates to this central data store. The central data store must then ensure all local data stores are kept synchronized. Essentially, this means that write actions that are performed must update both a local and remote data store, potentially hampering performance. This problem is similar to the problem of synchronizing local database caches, of which several schemes have been described in the literature [88, 64, 7, 76, 78]. In Chapter 8, this problem will be discussed in more detail and a trade-off analysis is described to support a decision support process to determine whether it is worthwhile to vertically integrate a code that executing a certain query mix.

7.5 Validation

To validate the effectiveness of our approach, we have performed experiments with the RUBBoS [74] and RUBiS [75] benchmarks. For each benchmark, instruction count measurements have been conducted on three versions of the code. The first instruction count is the original version of the code written in PHP and executed using Apache and MySQL. The second count is based on a vertically integrated code, that has been transformed by our vertical integration compiler using the procedure described in Section 7.3. The third count is the result of optimizing the code by hand, also reported in Section 7.2.

All experiments have been carried out on an Intel Core 2 Quad CPU (Q9450) clocked at 2.66 GHz with 4 GB of RAM. The software installation consists out of Ubuntu 10.04.3 LTS (64-bit), which comes with MySQL 5.1.41 and Apache 2.2.14. The instruction count measurements have been performed using the *oprofile* software, sampling the *INST_RETIRED* hardware performance counter present on the Intel Core 2 CPU.

The different PHP scripts, or components, that make up each benchmark have been benchmarked separately. For the instruction count measurement, each component has been executed multiple times, if applicable with different CGI arguments. This is to straighten out fluctuations, incorporate the effect of different CGI parameters and to collect enough samples for the profiler to produce meaningful results. Execution of a component is triggered by requesting the respective page with an HTTP client (e.g. `elinks -source`). The final result is the average number of instructions executed for a single execution of the component. *oprofile* reports instruction count samples per process and only the instructions executed of these processes that play a role in the generation of the page have been aggregated. So, other processes running on the system did not influence the measurements. The HTTP client has been excluded from the aggregate instruction counts.

7.5.1 Read-only Operations

Currently, our vertical integration compiler is able to transform the code of 7 PHP scripts in RUBBoS and of 9 PHP scripts in RUBiS. The realized instruction count reductions are shown in Figure 7.2. The bars on the left of the dashed line are the results for the RUBBoS benchmark components, the results on the right for RUBiS. PHP scripts that have been executed with different input parameters are marked with "*" and the average was taken of the results. The white bar indicates the percentage of instructions that has been eliminated, the gray the percentage of instructions that remain.

The results show that our current vertical integration compiler is able to realize a significant (43.0% to 87.0%) reduction in instructions for the different components of RUBBoS and up to 95.3% for the RUBiS components.

Three of the RUBBoS components and 4 of the RUBiS components have also been optimized by hand. The results of the hand-optimized codes are shown in Table 7.1. Note that even though our compiler is a work-in-progress prototype, it is already able to produce codes that achieve a performance close to hand-optimized codes for these benchmarks (differences in the range of only 0.4% to 5.3%). Continued development of the prototype will decrease the gap between automatically and hand optimized codes. Secondly, we notice that RUBBoS and RUBiS are quite simple codes, where the optimization techniques that become possible after incorporation of the query codes cannot be utilized to their fullest potential. For example, in Chapter 2 we showed that in other applications the integration opportunities were much more versatile, in particular in the phase after incorporation of the query codes, in the order of another 10%. Therefore, we believe that the results reported in this section form a lower-bound for results that can be achieved for more complicated applications that we plan to survey in the future.

7.5.2 Read/Write Operations

In this subsection, we present the reduction in instructions that can be obtained for read/write components. Results for automatic and hand optimized codes for 4 read/write components of the RUBBoS benchmark are shown in Figure 7.3. We

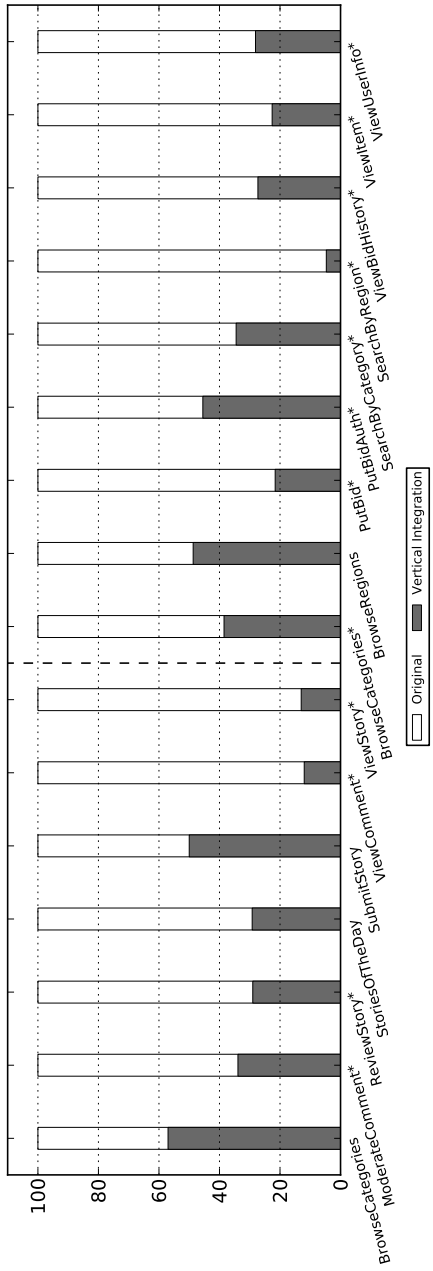


Figure 7.2: Percentage in instruction count reduction over the original Apache/MySQL execution for 7 of the RUBBoS benchmark components on the left and 9 of the RUBiS benchmark components on the right. Results realized by an automatic transformation of the code are shown. A "*" indicates an average is shown of performing experiments with different arbitrarily chosen input parameters.

Component	Automatic	Hand
RUBBoS StoriesOfTheDay	70.8%	76.1%
RUBBoS ViewComment	88.0%	93.3%
RUBBoS ViewStory	87.0%	92.7%
RUBiS SearchByCategory	65.6%	70.2%
RUBiS ViewBidHistory	72.7%	73.5%
RUBiS ViewItem	77.4%	77.8%
RUBiS ViewUserInfo	71.9%	74.9%

Table 7.1: Percentage in instruction reduction over the original Apache/MySQL execution for only these RUBBoS and RUBiS benchmark components that have been optimized by hand.

have selected these components from the RUBBoS benchmark, because they exhibit more interesting query mixes compared to the components found in RUBiS. The translation performed for the write operations is obtained by performing only an update of the local data instead of updating data in a remote DBMS. The results indicate that a reduction in the number of instructions executed is possible of around 70%.

Although notable speedups can be achieved with vertical integration of Read/Write components, frequent updates to local data stores is not a scalable solution for distributed deployments of the application. Therefore, we must consider the performance effect of distributing the updated data to the other nodes in the system. A number of methods to solve this problem will be described in Chapter 8. In general, for codes with many write actions, the cost of distributing the updates may not weigh up to the benefits attained by vertical integration. To quantify this, in Chapter 8 a trade-off analysis is presented to support automatic decision making whether or not a code with a certain query mix should be vertically integrated. In this chapter, we assume a setup where updates are immediately applied on the local data and are synchronously submitted to a main DBMS. Other setups are of course possible and a similar trade-off analysis can be carried out in these cases.

Using this methodology, we make a prediction of the speedup that can be achieved in query processing when a transition is made to query evaluation local to the application and additionally sending updated data to a central DBMS. In order to do so, the speedup is correlated to different ratios of *remote* writes against *local* writes. That is, we can control the amount of write actions that are also applied at a central location. For writes that are only applied locally, this has as a possible consequence that components that require this data can only be executed on one particular host. This way, a distributed deployment of a web application can be tuned in different ways.

The speedup predictions as described above can be displayed graphically in a contour plot, see Figure 7.4. From the figure, the predicted speedup can be read for a given component and remote/local write ratio. Note that all speedups reported in the figure are above 1.0. So, even if all write actions are applied remotely as well, it is predicted that an overall speedup can be achieved. This is due to the fact that every component performs at least a single read action. In case of *Store-*

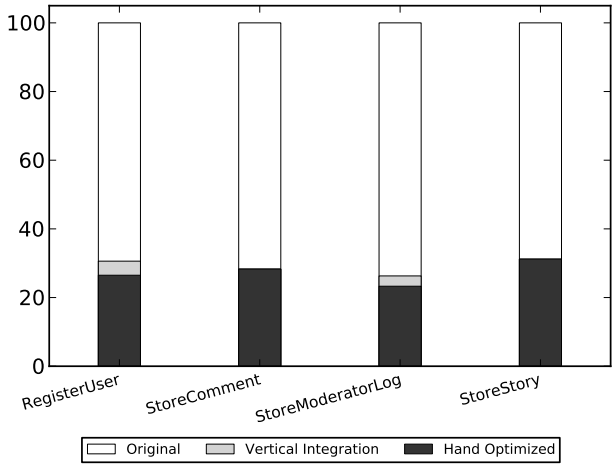


Figure 7.3: Percentage in instruction reduction over the original Apache/MySQL execution for 4 R/W components of the RUBBoS benchmark. Results realized by an automatic transformation of the code and hand-optimized code are shown.

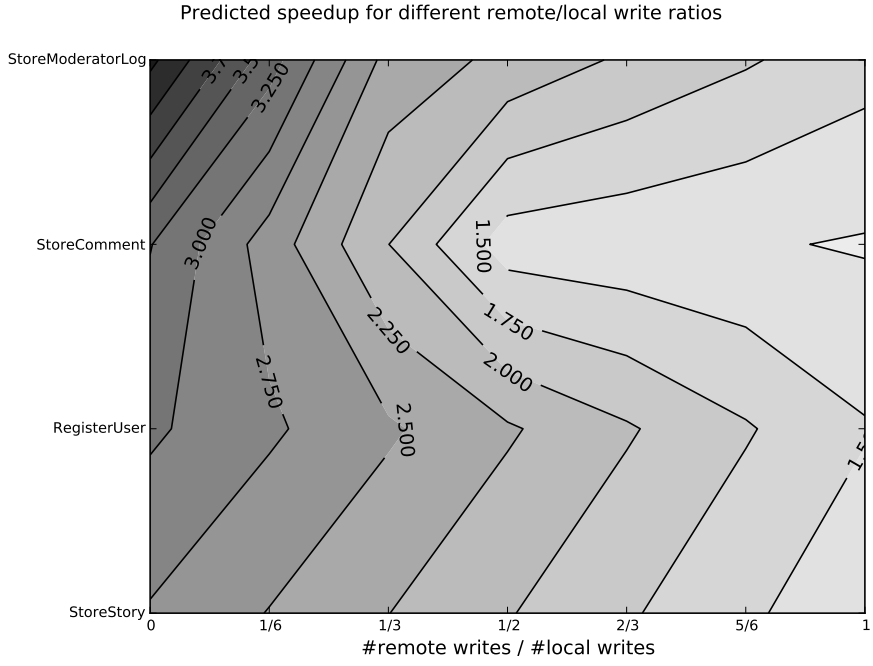


Figure 7.4: Contour plot of the predicted speedups (contour lines) for different read/write benchmark components and remote/local write ratios.

ModeratorLog a speedup of a factor of 4 can be attained if all write actions are only executed locally. In case all actions are also performed remotely, still a speedup between a factor 1.5 and 1.75 is possible. From this contour plot can be seen that if the ratio is 0.5, meaning that the number of remote writes is one third of the total writes, that speedups range from 1.5 to 2.25, with *StoreComment* being most sensitive.

7.6 Further Optimizations

In the previous section, we have described that up to 95% of the instructions executed by database applications can be eliminated. As has been discussed in the Introduction of this chapter, a reduction of the number of executed instructions has a direct impact on the performance of the application.

Figure 7.5 shows the percentage reduction of the page generation time of the surveyed benchmarks realized by the automatic code transformation performed by our compiler. The page generation times were measured by storing time stamps at the start and end of the various PHP scripts and computing the time elapsed. Note that the percentage shown only reflects the reduction in page generation time. This excludes any speedup in the stages before execution of the PHP script has started, such as parsing the PHP script and generating executable code that is done by the PHP module in the Apache web server. Due to the nature of our measurements using *oprofile*, these were included in the instruction count measurements reported in Section 7.5. Due to the use of HipHop for PHP, the PHP parsing and code generation is no longer done at runtime, so the total reduction of execution time is larger.

The results show that for the majority of the benchmark components the page generation time is reduced by over 80%. For a number of pages, the page generation time is reduced by around 95%.

These achieved speedups are a direct result of the instruction count reduction. To accomplish this reduction in instructions, the layers used to compose the database application had to be broken down. For example, the use of a modular interface to the DBMS has been removed and replaced with query evaluation within the same process. This has reduced the number of executions to be executed, as data no longer has to be boxed for transfer to and from a DBMS. Important is that the elimination of the use of a DBMS has as a significant side effect that time is no longer lost by transferring data between two processes on the system and context switching overhead. So, even though the expected total reduction of execution time is larger than reported, the reported reductions in page generation time are still larger than the reported reduction in instruction count.

Note that all the results reported in this chapter up till now, have been obtained without any advanced compiler optimizations, like loop transformations and code optimizations, see the previous chapters. Due to the use of the *forelem* intermediate representation, both the application logic as well as the database queries are represented as loops. On these loop structures, a compiler can perform traditional, sophisticated loop optimization, resulting in a further substantial improvement in performance. Additionally, other effective optimizing compiler

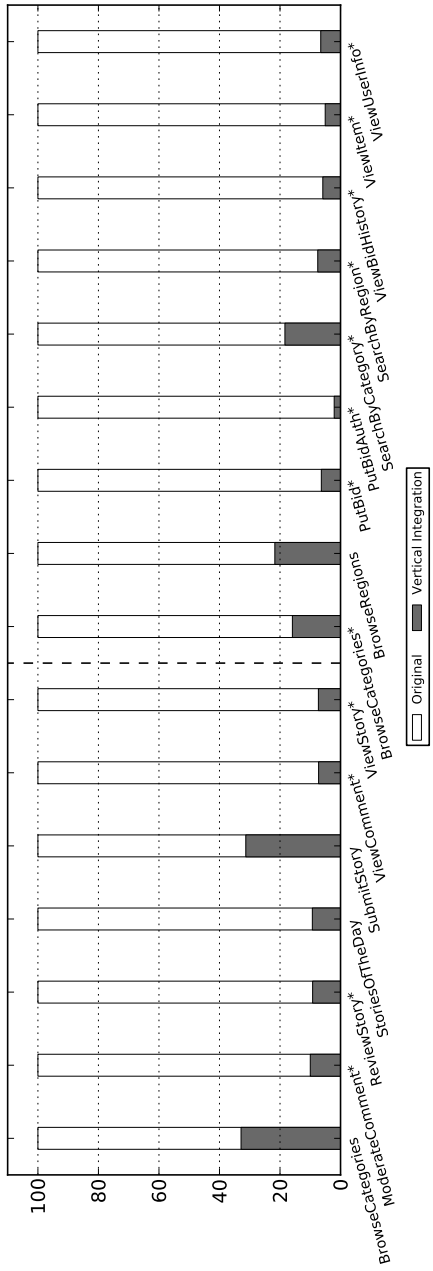


Figure 7.5: Percentage in page generation time reduction over the original Apache/MySQL execution for 7 of the RUBBoS benchmark components on the left and 9 of the RUBiS benchmark components on the right. Results realized by an automatic transformation of the code are shown. A "*" indicates an average is shown of performing experiments with different arbitrarily chosen input parameters.

techniques can be applied to further optimize the code, such as optimizations for more efficient data structures, register allocation, cache usage and data prefetching. For an example of the transformations that can be applied after the DBMS API has been replaced with *forelem* loops, see Chapter 3 (Section 3.4).

We have demonstrated in this chapter that an automatic reduction of a database application to its essence is feasible. This uncovers a significant number of unnecessary instructions, which are automatically eliminated, having a direct beneficial impact on the application's performance. The fact that the application program logic and its queries are expressed in a common intermediate representation, forms a basis for the definition of code transformation specifically for database applications.

7.7 Related Work

Cheung et al. describe a system, StatusQuo, to optimize the performance of database applications written in Java accessing a database through JDBC or Hibernate by considering both the application code as well as the queries [22]. Similar to our approach they state that the hard separation between the application and database code often results in applications with suboptimal performance. The system is capable of automatically partitioning the database application into a Java and SQL code, for optimal performance. To accomplish this, it may rewrite SQL into Java code, or vice versa. Whereas StatusQuo translates imperative code into a declarative form, our system translates declarative code into an imperative form. Furthermore, we accomplish a significant reduction in the number of instructions to be executed.

Holistic transformations for web applications are proposed in [66, 37], where transformations are performed on both the application code and the database queries performed by the application code. The division between application and database codes remains in place however. A similar approach for database applications written in Java is discussed in [21].

7.8 Conclusions

In this chapter, we have presented the results of our prototype compiler that is able to automatically reduce the number of instructions executed by database applications by up to 95%. These results have a direct effect on the performance of the application, with page generation times also being reduced to up to 5% of the original page generation times.

Next to these significant results, we have demonstrated that it is feasible to automatically translate applications to a substantially reduced form. This is accomplished through the vertical integration methodology. Our current prototype compiler is capable of achieving a performance near that of the hand-optimized codes. In future work, we will further improve our compiler technology to match and go even beyond the performance of the hand-optimized codes. Furthermore, we are working on developing novel compiler optimizations that will further improve the performance of queries translated using the *forelem* framework.

The current prototype of the reduction process is capable of processing PHP codes by a translation to C++ code using the HipHop for PHP project. Many other programming languages and frameworks are also used to develop database applications. The general nature of the *forelem* framework does not restrict its usage to the C and C++ programming languages. As such, our prototype can be extended to be able to process applications written in other programming languages.

CHAPTER 8

A Trade-Off Analysis For Locally Cached Database Systems

8.1 Introduction

As can be seen in Chapter 7, our approach to reduce database applications to their essence calls for exploiting locally cached data. Locally cached databases are in common use and different strategies have been described in the literature [88, 64, 7, 76, 78]. Many strategies work by caching (part of) the data set to satisfy all, or the majority, of the read operations locally. To handle write actions, however, these methods commonly work by immediately forwarding write actions to the main DBMS and by relying on the main DBMS to propagate updated data to the local caches. Solutions for different caching schemes are often evaluated using full transactional workloads, which in the case of e-business processing are often read-dominant, as these applications have a high browse-to-order ratio [64]. As a result of table caching, many of the read queries of these workloads do not have to be passed to the main server, so it is clear that caching can noticeably improve the performance in such cases [64, 7].

Contrary, if a workload is not read-dominant, the advantage of the local execution of read queries may not weigh up to the amount of time taken by the remote execution of write queries. In fact, the overhead introduced by updating local copies of the data next to, or in response to, write actions may become a problem. A similar problem occurs when full local copies of a database are cached so that the data processing and data retrieval codes can be combined in the same process. These local copies must be kept synchronized across multiple machines. Evidently, this scales well for workloads that are heavily read dominant. For other workloads, an analysis is required to determine whether the local caching of a database will be beneficial, or that parts of this local cache are better served from a remote DBMS. To our knowledge, a framework to perform such a trade-off analysis is not described in the literature as of yet. In this chapter we introduce a

trade-off analysis for deciding whether or not the introduction of a local database will be beneficial for a given set of query mixes.

The framework works by determining significant computational parameters for local and remote query execution. Sequences of queries can be expressed in terms of these parameters and these parameters in turn can be used to predict the computational load of this query mix. In this chapter, the caching scheme that is considered is one where writes to the local database must be accompanied by writes to the remote, or master, database. For the sake of simplicity, the cost of different schemes of propagating writes and updating local caches is not considered. Note that it is not our intention to predict actual performance of locally cached database systems, but rather to provide an analysis for deciding whether a locally cached database solution will have benefits over solely executing queries on a remote DBMS.

The significant computational parameters within this framework can be determined via two approaches. The first approach uses a separate benchmark to approximate the computational load cost of different database operations. For the second method, a database application is adapted such that the execution time of the different query mixes can be measured. From these results, the values for the separate parameters are computed.

To illustrate our methodology, we apply our method to a straightforward example. This example is the RUBBoS benchmark [74], which models a web application written in the PHP language and using MySQL as DBMS. Different query mixes performed by this benchmark are used to collect measurements required for the second approximation method and to validate our two proposed methods. The use of the performance models are demonstrated on two different architectures.

The chapter is organized as follows: in Section 8.2 the significant computational parameters are discussed. Section 8.3 introduces and describes the experimental setup. Section 8.4 describes the first method to determine the significant computational parameters and the second method is described in Section 8.5. In Section 8.6 the trade-offs are analyzed for deciding whether to offload computational load from a main server using predictions generated with the framework. The conclusions are presented in Section 8.7.

8.2 Significant Parameters

For estimating the computational load, approximate cost figures for different local as well as remote operations are needed. The approximate cost of the local read, insert and update operations will be referred to as R_l, I_l and U_l respectively[1]. For remote operations it has been observed that when a number of SQL statements are executed operating on a same table (which fits in main memory), the first SQL statement that is executed after opening the connection to the MySQL server is more expensive than subsequent statements operating on the same table. Therefore, when a sequence of remote operations is performed on the same table

[1]Note that the delete operation is not explicitly discussed in this chapter, the methods presented can be easily extended to accommodate additional operations.

(regardless of type), the first of these operations is marked as a "high" cost operation and the subsequent operations as "low" cost. High cost remote operations will be referred to as R_{rh} (for read, remote high cost), I_{rh} (for insert, remote high cost) and U_{rh}. The low cost remote operations are referred to as R_{rl}, I_{rl} and U_{rl}. Because the performance model is to be used for a class of applications that typically open a connection, perform a number of queries and close the connection, the high cost operations cannot be amortized over subsequent operations.

There is one exception to the above described rules. If two similar operations on different tables are performed subsequently, then the second of these operations is to be counted as low cost. For example, if a select on table A is immediately followed by a select on table B, then the select on table B is counted as low cost, while according to the general rule it should have been counted as high cost. This exception is needed because there is empirical evidence that the cost for the second select is significantly lower than for the first select, even though the select is performed on a different table.

There is empirical evidence as to why this exception is required. We have timed the average execution time of the separate queries in the *StoreModeratorLog* benchmark in the *Remote* configuration (see Section 8.3). The results, in microseconds, are in Table 8.1. For reads and inserts, the top 3 and bottom 3 results out of 28 measurements were eliminated and the average is taken. For update operations, the top and bottom 2 results are eliminated out of 16 total measurements. Behind each result, the minimum and maximum measurements (after elimination of the top and bottom) are given within brackets. We observe from the table that the cost for the second select are significantly lower than for the first select, even though the select is performed on a different table.

	Average Execution Time	
Query	**Core 2**	**Core i7**
select A	171 [165, 177]	149 [144, 160]
select B	82 [78, 86]	77 [75, 79]
update A	99 [98, 100]	93 [91, 97]
update B	93 [87, 100]	80 [78, 81]
select B	66 [62, 69]	65 [63, 68]
select A	65 [63, 68]	64 [62, 62]
insert C	79 [71, 89]	72 [66, 79]

Table 8.1: Execution time in microseconds of the separate queries in the *StoreModeratorLog* benchmark in the *Remote* configuration.

If the insertion of such exceptions were to be avoided, more parameters have to be introduced. Apart from distinguishing only high and low cost operations based on whether the table has been used in a query before in the active connection, distinctions can also be made in whether it is the very first query in the active connection, what kind of index is used for the table that is being queried (normal or primary), etc. An important consequence of having too many parameters in a model is that all possible predictions can be obtained simply by tweaking with all parameters. This renders the model useless. To avoid having too many parameters we have deliberately chosen to not take further effects into account.

Finally, next to the parameters for the approximate cost of the different operations, there is one parameter for the "base time". The base time includes the time required to execute the benchmark code which is submitting the queries, as well as the time required to setup the connection to the remote database server. This time is always measured separately for each respective benchmark. So, if predictions are to be made for a certain benchmark, its base time must be measured beforehand.

Note that for benchmarks only performing local queries, the connection setup time is subtracted from the base time. The connection setup time can be determined by taking the average difference of the page generation time of experiments only performing local queries that do set up a connection to a remote database server and of experiments that do not setup a connection. It is also possible to define the base time as solely the execution time of the benchmark code. However, given the fact that we had more configurations which include both execution time and connection setup time, we have chosen to define the base time as the accumulation of these times.

8.3 Experimental Setup

Four components (PHP scripts) from the *RUBBoS* [74] benchmark will be used for demonstrating the trade-off analysis. By exploiting the different read/write characteristics of these four components the significant parameters will be determined with one of the components and the performance model will be validated against the remaining three components. The RUBBoS benchmark was developed by a collaboration between RICE University and INRIA and models a typical bulletin board system or news website with possibility to post comments. The PHP-version of RUBBoS has been used and this code base was translated to C++ code using the HipHop for PHP project [42]. The C++ code base facilitates the merger of the application code processing the data and the data retrieval code. This is done by embedding a generic local cache, based on flat C arrays, in the application code. This is of course a gross oversimplification of the actual implementations of locally cached databases, but it ensures that we do not penalize particular implementation choices made for these locally cached databases. The end result is compiled to a native executable.

The resulting executables have been benchmarked on two generic Linux systems. The first system is based on an Intel Core 2 Quad CPU (Q9450) clocked at 2.66 GHz with 4 GB of RAM. The software installation consists out of Ubuntu 10.04.3 LTS (64-bit), which comes with MySQL 5.1.41.

The second system is based on an Intel Core i7 CPU 2820QM clocked at 2.30 GHz with 8 GB of RAM. This machine was running Ubuntu 11.10 (64-bit), which comes with MySQL 5.1.61. Similar configuration files were used as those for the first system. In order to obtain consistent results the system was configured to disallow the system from entering a sleep mode beyond C1 and the clock frequency was locked at 2.30 GHz.

To be able to carefully analyze the trade-offs involved in offloading computational load of the main server for different query mixes, the page generation times

were collected of web pages generated by performing different query mixes. The page generation time is defined as the difference between the time the first line of the (translated) PHP code started execution until the time the last line of the PHP code is executed. This time does not include any initial startup cost for serving the request for the web page. The web pages that were selected from RUBBoS are:

- *StoreModeratorLog*, which performs: two selects, conditionally two updates, two selects and one insert.
- *RegisterUser*, which performs: one select, one insert and one select.
- *StoreComment*, which performs: one select, one insert and one update.
- *StoreStory*, which performs: one select, one insert.

With the four mentioned components, experiments have been performed in different configurations. The results of these experiments are used in Sections 8.4 and 8.5 to create and to validate the performance model. The configurations range from performing all queries "remotely" in the MySQL server to performing all queries "locally" in a generic data store. Note that while query execution in MySQL is referred to as "remote" execution, the MySQL server is running on the same system that performs the local operations. With "remote", it is specified that the request has to go out of process and a connection has to be set up with a MySQL server. The exact configurations are as follows:

- *Remote*, all queries are executed through MySQL.
- *L+R*, all queries are executed locally and all write queries (insert, update and delete) are also executed through MySQL.
- *L+C*, all queries are executed locally and a connection is setup to the MySQL server (but no query is performed through MySQL). This result is used to approximate the overhead of setting up a connection to the MySQL DBMS.
- *Local*, all queries are only executed locally.
- *Base*, no query is executed, however a connection is setup to the MySQL server. This configuration is used to determine the base overhead of a benchmark by comparing the result with the result for *L+C*.

For the configurations in which queries are performed locally, such as *L+R* and *Local*, the generated C++ code was modified by embedding a generic local cache. This enables a clear estimation of the minimum computational cost of local database operations. Remote updates are considered to be part of the updates which are performed locally. That is, the local program will not continue until the updated data has been committed in the main DBMS.

Specific implementations of local data stores will differ from this generic implementation. For example, instead of evaluating a query which processes a write action locally, an implementation can choose to simply forward the query to the main DBMS and wait for an update to be propagated. Other implementations of local data stores, such as the Oracle In-Memory Database Cache product [76],

are capable of both processing the remote updates as part of the local transaction (synchronously) and processing the remote updates asynchronously outside of the transaction. The latter is said to yield significant performance benefits. However, whether synchronous or asynchronous updates should be used, depends on the application and its requirements.

8.4 Method 1: Benchmarking

We devised a separate benchmark for determining the different approximate computational load costs for both local and remote operations. This benchmark operates on a single table with a single table column. No primary key was specified and an index has been created on this single column after creation of the table. To determine the cost parameters, the execution time was measured of each of the operations while operating on different table sizes. The table sizes ranged from 1 tuple up to 671088640 ($2^{27} + 8 \times 2^{26}$) tuples. When the benchmark is performed, both architectures were configured to only use 2GB of RAM in total. This way out-of-memory performance can be studied without generating very large data sets. The different operations have been measured as follows:

- **Read** A *select* statement is performed, which in the *where* clause compares the value of the table column with a randomly chosen value. Because the table was populated with unique numbers, this always results in a result set of equal size (1 tuple). Each experiment performs 5 statements and the experiment is repeated 30 times. All execution times are averaged.

- **Update** An *update* statement is performed, updating a randomly chosen value to the same value. Because the table was populated with unique numbers, similar to read, this always results in an update set of equal size. Each experiment performs 5 statements and the experiment is repeated 30 times. All execution times are averaged.

- **Insert** Simple *insert* operations are performed. This experiment is repeated 5 times, because there are not many fluctuations due to the absence of random table access. Each experiment performs 5 insert operations.

For the remote operations the difference between the "high" cost and "low" cost operations must be measured. This difference was measured by taking the first value as "high" cost and the last value as "low" cost. The average of high cost and low cost operations is taken over all performed experiments (which is for read and update 30, so this gives 30 experiments with 5 query evaluations each). When randomly generated values are used, the local experiment will pick a random value for each run of the query while the remote experiment picks a random value for each experiment (and the value is thus used in 5 statements).

The results of the benchmark runs with remote operations for the Core 2 and Core i7 systems are shown in the top row of Figure 8.1. All execution times are in microseconds. Due to the use of an index, the variance in the results for the remote operations is low. When the same experiment is performed on a table without an

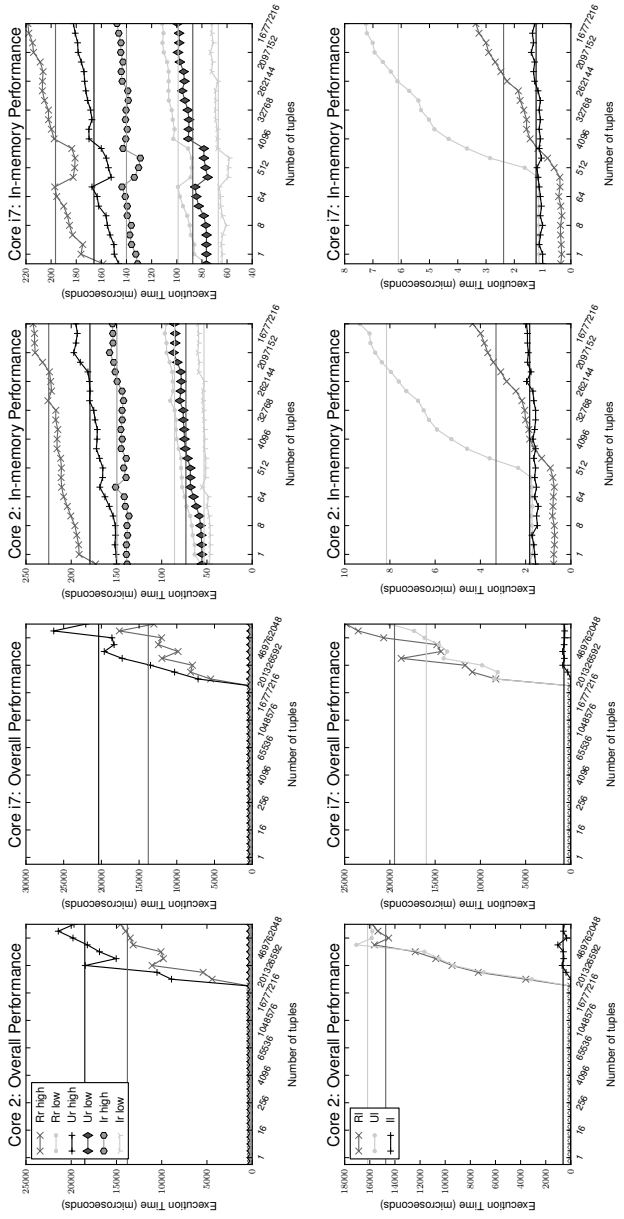


Figure 8.1: Overview of execution time of remote operations on the top row, local operations on the bottom row. Overall Performance depicts the measured query execution time for increasing table size, expressed in number of tuples, up to a table that no longer fits in main memory. In-memory Performance zooms in on the range of table sizes that fits in main memory. Straight lines indicate the identified plateaus.

index, a linear relation is indeed observed between the execution time of the query and the size of the table.

It is important to realize that the experimental setup of this benchmark implies that the high and low cost parameters serve a different purpose for out-of-memory operations. The difference seen between high and low cost operations for in-memory table sizes is too small to be noticed in out-of-memory operations. Because the same random value is used for the set of 5 query runs from which the high and low cost values are taken, high and low in fact stand for cold cache and warm cache performance. The use of high and low cost for out-of-memory operations in a computation depends on whether spatial locality is exploited by the query mix. Therefore, the exception posed in Section 8.2 does not apply to out-of-memory operations.

In the bottom row of Figure 8.1 the results for the local operations are shown. For both local and remote operations it can be clearly seen where the database table runs out of memory. In the graph for the local operations we observe that for very small table sizes, the query execution times stay constant. This is likely caused by the fact that the entire table fits in CPU cache if the table size is sufficiently small.

Because there is a very large gap between the query execution times when the table fits in system memory and when it does not, different sets of parameters are determined for different table sizes. In the graphs, two plateaus are identified, one for in-memory tables and one for out-of-memory tables. The plateaus were determined by taking the average execution speed of each type of query for a table size range were performance is similar and are plotted in the graphs as straight lines.

The two largest tables in the RUBBoS data set consist of 200000 and 500001 tuples. For both architectures, this corresponds with an in-memory table size. We thus use the parameter set for an in-memory table size. The parameters are shown in microseconds in Table 8.2.

Core 2	$U_l = 8.15$	$I_l = 1.82$	$R_l = 3.31$
	$U_{rh} = 179.30$	$I_{rh} = 149.52$	$R_{rh} = 224.88$
	$U_{rl} = 73.25$	$I_{rl} = 53.25$	$R_{rl} = 85.82$
Core i7	$U_l = 6.10$	$I_l = 1.23$	$R_l = 2.38$
	$U_{rh} = 165.89$	$I_{rh} = 140.03$	$R_{rh} = 196.49$
	$U_{rl} = 87.28$	$I_{rl} = 66.95$	$R_{rl} = 98.92$

Table 8.2: Determined significant parameters, in microseconds, for in-memory table sizes.

These parameters will now be validated against the 6 query mixes taken from the RUBBoS benchmark. From this benchmark, three components were used to generate the query mixes: *RegisterUser* ($R + I + R$), *StoreComment* ($R + I + U$, $R + U$, $I + U$) and *StoreStory* ($R + I$, I). For two of the components, the additional query mixes have been generated by slightly modifying the code, for example to bypass a query used for authentication. In general, experiments were performed on a warmed up server and 1 or 2 top and bottom results were eliminated before averaging the results.

The parameters are validated by comparing the predictions, which are obtained by filling the parameters into the formulas, to the actual measured page generation times. Table 8.3 shows the deviation between the predicted and measured execution time for the *RegisterUser*, *StoreComment* and *StoreStory* benchmarks. Note that the *exec* and *Base* values amongst benchmarks are different, the one corresponding to the benchmark is used. The *exec* values are found by correcting the *Base* values respective to the benchmark for the connection overhead. The connection overhead is determined by subtracting the measured value for *Local* from *L+C*. In case of *StoreComment* and *StoreStory* multiple experiments are available, the connection overhead is computed for all experiments and the average is taken.

	Core 2 Deviation	**Core i7 Deviation**
	***Local* configuration**	
exec+$R_l + I_l + R_l$	-40.37%	-54.10%
exec+$R_l + I_l + U_l$	-11.53%	-6.52%
exec+$R_l + U_l$	6.10%	8.33%
exec+$R_l + I_l$	8.41%	5.98%
exec+$R_l + I_l$	-35.62%	-39.84%
exec+I_l	-30.87%	-30.13%
	***Remote* configuration**	
Base+$R_{rh} + I_{rl} + R_{rl}$	-11.38%	2.37%
Base+$R_{rh} + I_{rh} + U_{rl}$	12.39%	27.00%
Base+$R_{rh} + U_{rh}$	34.04%	23.90%
Base+$I_{rh} + U_{rl}$	-11.47%	10.82%
Base+$R_{rh} + I_{rh}$	23.23%	20.87%
Base+I_{rh}	-2.22%	24.16%
	***L+R* configuration**	
Base+$R_l + I_{rh} + I_l + R_l$	-7.29%	9.75%
Base+$R_l + I_{rh} + I_l + U_{rl} + U_l$	-16.06%	3.66%
Base+$R_l + U_{rh} + U_l$	10.84%	1.90%
Base+$I_{rh} + I_l + U_{rl} + U_l$	-11.31%	12.70%
Base+$R_l + I_{rh} + I_l$	-10.71%	12.59%
Base+$I_{rh} + I_l$	-9.01%	10.35%

Table 8.3: Deviation of the predicted results, using the parameters determined with the first approximation method, and actual results.

Observe that for both systems, the predictions for *RegisterUser* and *StoreStory* are off by at least 30%. The query mixes all have in common that they contain an insert. It is likely that the value determined for the I_l parameter by the separate benchmark is too small. This could be caused by the fact that more time is spent updating the several indexes on these tables, compared to the single index that is updated in the separate benchmark.

Table 8.3 also presents the results for the validation of the remote operation parameters under the heading *Remote*. Four out of six of the Core 2 results are within a 12.5% margin. The Core i7 results show a different picture, all predictions are higher than the actual execution time, with the majority of the predictions

being at least 20% too high. A possible explanation for this overestimate is that the values of the parameters are too high in relation to the sizes of the tables in the used data set. One way to improve the accuracy of these parameters is to define plateaus to be lines with a slight slope instead of a straight line. Instead of a parameter value, we will obtain parameter functions linear to the table size.

The results for the validation of performing all queries locally and performing all write queries remotely as well are shown in Table 8.3 under the heading *L+R*. It was observed that the predictions for the Core 2 system are for the majority within a 11% margin. For the Core i7 system, the results appear to be slightly better with half of the predictions in a 10% margin and all predictions within a 12.7% margin.

8.5 Method 2: In vivo

With the method described in the previous section, execution time is measured using tables containing a single column. What is measured is in fact kernel performance. A drawback of this method is that kernel performance is used to predict application performance. To alleviate this drawback, a second, *in vivo*, method is introduced where an existing benchmark is used to determine the significant parameters as opposed to using a separate benchmark. The selected benchmark should be modified such that it can be run with different query mixes. Ideally, the selected benchmark already performs each of the read, insert and update operations such that the different query mixes can be created by simply disabling subsets of the queries in the benchmark. To obtain results for both local and remote operations, the benchmark is run in both the *L+C* and *Remote* configurations.

From the results, the parameters are computed by setting up a linear system of equations of the query mixes and the resulting execution times (subtracted by the measured *Base* time for this benchmark, such that the execution time of only the queries remains). The rank of the system of equations should be equal to the number of unknowns and the number of equations in the system must be larger than the rank to straighten out perturbations arising from the measured execution times. When the number of specified equations is larger than the matrix' rank, the system can be solved using the least squares method. To determine the local cost parameters, a system of 5 different equations is set up and solved. The resulting 3 values for the system's variables are taken as parameters.

The remote cost parameters consist out of low cost and high cost parameters. To find these parameters a system of 8 different equations is used. Of these 8 equations, 3 must reflect query mixes that determine the high cost parameters, i.e. these are query mixes that only perform 1 read, 1 update or 1 insert. The selected benchmark can be easily modified to only allow for execution of one of these queries and to disable the other queries. The inclusion of these 3 query mixes counters the tendency of the least squares method to move the values of the low and high cost parameters towards each other. The high cost parameters are taken from the 3 specific query mixes. The low cost parameters are obtained from the solution of the system of equations.

We will demonstrate this method using the RUBBoS benchmark. The *StoreModeratorLog* component of the benchmark was modified to perform the neces-

sary different query mixes to be able to determine all parameters. The additional query mixes required for determining the remote parameters have only been performed with the *Remote* configurations. Because the RUBBoS data set in use is categorized as small, the result will be a parameter set for in-memory table sizes. With each query mix 12 experiments have been performed on a warmed-up server, eliminating the top 2 and bottom 2 results and averaging the remaining results. Using these results the linear systems of equations are set up to determine the approximate cost for each of the operations. The cost of remote parameters is approximated using the *Remote* results, while the *L+C* results are used to approximate the local cost. From these results, the *Base* has been subtracted before the numbers are entered into the system of equations. This is done to remove the base time for the script execution as well as the time required to set up a connection to the MySQL server. The resulting approximations are listed in Table 8.4.

Core 2	$U_l = 7.6$	$I_l = 12.75$	$R_l = 9.4938$
	$U_{rh} = 181.0$	$I_{rh} = 141.0$	$R_{rh} = 192.0$
	$U_{rl} = 88.3477$	$I_{rl} = 35.8591$	$R_{rl} = 66.7731$
Core i7	$U_l = 1.925$	$I_l = 9.875$	$R_l = 5.9906$
	$U_{rh} = 154.1$	$I_{rh} = 109.7$	$R_{rh} = 162.9$
	$U_{rl} = 86.4873$	$I_{rl} = 22.3849$	$R_{rl} = 67.7115$

Table 8.4: Significant parameters as determined using the second approximation method.

Similar to the discussion in Section 8.4, these obtained parameters will be validated by considering the deviation between the predicted and measured execution time. The results are shown in Table 8.5. For both platforms, one real outlier is observed. The other results are all within acceptable margin from the measured result. The Core i7 results are within a smaller margin from the measured results compared to the Core 2 results. Overall, the predictions for both platforms are clearly better than the predictions by the first method, where four out of six predictions were off by at least 30%. This is most likely caused by a small I_l parameter. Note, that the I_l parameter value found by the second method is substantially greater than the one found by the first method. For example, for the Core 2 system the first method found a value of 1.82 while the second method found a value of 12.75.

Under the *Remote* heading in Table 8.5, the validation of the predictions for the remote operations produced by the second method are shown. When the results for the Core i7 system are compared to the results for the first method (Table 8.3), the consistent overestimates are no longer present. Instead, the majority of the results is within a margin of approximately 12.5%. The Core 2 results show a similar trend with the majority of the results within a 13.25% margin, and two real outliers. Compared to the first method, the results for the first and second method are on the same footing for the Core 2 system.

Finally, the predictions for performing all queries locally and performing the write queries remotely are shown in Table 8.5 under the *L+R* heading. The predictions for the Core 2 system are quite good, with all predictions being within a 13.3% margin from the actual measured result and 4 out of 6 within 8.25%. This is better than the predictions that resulted from the first method. The sec-

	Core 2 Deviation	Core i7 Deviation
	***Local* configuration**	
exec+$R_l + I_l + R_l$	-9.19%	-31.34%
exec+$R_l + I_l + U_l$	-13.76%	7.38%
exec+$R_l + U_l$	16.75%	7.17%
exec+$R_l + I_l$	29.00%	15.10%
exec+$R_l + I_l$	-11.69%	-4.83%
exec+I_l	-13.19%	6.56%
	***Remote* configuration**	
Base+$R_{rh} + I_{rl} + R_{rl}$	-23.50%	-20.40%
Base+$R_{rh} + I_{rh} + U_{rl}$	7.36%	1.81%
Base+$R_{rh} + U_{rh}$	26.33%	18.73%
Base+$I_{rh} + U_{rl}$	-9.83%	-12.60%
Base+$R_{rh} + I_{rh}$	13.25%	-6.96%
Base+I_{rh}	-5.10%	-8.87%
	***L+R* configuration**	
Base+$R_l + I_{rh} + I_l + R_l$	-2.70%	13.71%
Base+$R_l + I_{rh} + I_l + U_{rl} + U_l$	-10.84%	-15.77%
Base+$R_l + U_{rh} + U_l$	13.32%	6.83%
Base+$I_{rh} + I_l + U_{rl} + U_l$	-7.23%	-9.52%
Base+$R_l + I_{rh} + I_l$	-8.04%	12.84%
Base+$I_{rh} + I_l$	-8.22%	-15.98%

Table 8.5: Deviation of the predicted results, using the parameters determined with the second approximation method, and actual results.

ond method, however, yields slightly worse results for the Core i7 system. All results are within a 16% and 2 results are within a 9.5% margin.

8.6 Analysis of Trade Offs

In the preceding sections, the significant computational parameters were derived and a performance model was determined. Using these results, it will be shown how an analysis of the trade-offs for deciding whether to offload computational load from a main server by local caching can be performed.

For predictions for remote operations and combined local and remote operations, the second method, *in vivo*, has been shown to produce slightly better results than the first method. In order to carry out the analysis, the predictions used do not have to be accurate within a few percent. We deem the predictions produced with the *in vivo* method to be accurate enough, because with a 16% margin the prediction method is more than able to show trends when offloading the computational load is beneficial. Additionally, if is also considered that the measured results used for validation, and for parameter computation in case of the *in vivo* method, have a 4% to 5% error margin on average, a 16% error for predictions is quite good.

As an example, for different query mixes the speedup has been computed of fully remote execution of all queries versus local execution of reads and local as well as remote execution of write actions. Each kind of operation is considered to be performed on a separate table, i.e. all reads are performed on one table, all inserts on another and all updates on yet another. The first of each operation is a high cost one and the subsequent of each operation is a low cost one. Each prediction includes the measured time required for setting up a connection to the MySQL server. For the connection time the average of the connection overheads measured in the benchmarks was used.

Figure 8.2 shows plots for a single update and a varying amount of reads and inserts, a single insert and a varying amount of updates and reads and a single read and a varying amount of inserts and updates. Plots are shown for the parameters found for the Core 2 system in the top row and the parameters found for the Core i7 system at the bottom. For our analysis we are interested in the areas with the largest speedup, because these areas indicate where offloading of computational load can have most benefits. Also of interest are areas with a speedup below 1.0, these areas indicate a slowdown meaning that a local cache is detriment for this particular workload.

Considering the leftmost plots of Figure 8.2, where the number of reads and inserts are varied, the effect of an increasing number of reads can be clearly seen when the number of inserts is kept small: the achieved speedup increases. If the number of reads is kept at zero, one or two and the number of inserts is increased, a slowdown is predicted. Notice the area between the y-axis and the 1.0 contour line, where the values are below 1.0. Because for these cases almost all operations are also performed remotely, there is no advantage to local caching for such query mixes. The plots in the middle column, depicting varying numbers of reads and updates, are similar.

From the same plots it can be observed for the read-insert case that when 10 inserts are performed with the Core 2 system, approximately 18 reads are needed to achieve a performance improvement of a factor of 2. For the Core i7 system approximately 13 reads are required against 10 inserts to have a factor of 2 speedup. Less reads are required compared to the Core 2, because the speedup of the read operations is significantly higher for the Core i7 system. When the derived parameters for the "low cost" remote read are considered, it can be seen that the speedup of going from a remote read to a local read is 11.3 for the Core i7 system and 7.0 for the Core 2 system.

A different situation is observed in the plots in the rightmost column of Figure 8.2, depicting varying numbers of updates and inserts and a single read. The speedup is only predicted to be above a factor 1.5 when a single insert or update is performed. Otherwise a speedup between 1.0 and 1.5 is seen up to 15 inserts, 20 updates, or a mix thereof. From this can be deduced that with only a single read that is converted from a remote read ("high" cost, because only a single read is done) to a local read, 15 inserts, 20 updates or a mix of these can be performed while a speedup, albeit meager, is obtained. If more inserts and updates are performed, a slowdown is seen. For the Core i7 system the area for which a small speedup is obtained is larger, again due to the larger speedup that was found for the conversion from a remote read to a local read. In this case, conversion

from a remote "high" cost read to a local read should be considered, resulting in a speedup of 20.2 for the Core 2 system and 27.2 for the Core i7 system.

The small area at the bottom left corner in which a speedup can be obtained deserves further investigation. Figure 8.3 zooms in on this area, only plotting up to 5 inserts and updates on the axes. A plot is shown with 15 reads (plots with different amounts of reads have been omitted due to space restrictions). From the plot can be concluded that when the number of reads is around 15, it is always possible to obtain a factor of 2 speedup when the mix contains 5 inserts, 5 updates or a mix thereof. For the same case, but with 20 reads, a speedup of approximately 2.5 for the Core 2 system and close to 3.0 for the Core i7 system is predicted. When 2 reads and updates are considered instead, it is concluded from the plot that a speedup of 4.0 cannot be obtained for the Core 2, but with the Core i7 system this is possible. This conclusion is in line with the left two columns of plots seen in Figure 8.2. In these plots, a speedup of 4.0 is not seen for the Core 2 system and only seen in a small region for the Core i7.

8.7 Conclusions

In this chapter a framework was introduced to support a trade-off analysis to help decide whether introduction of a local cache of a database reduces the total computational load. This framework uses two methods to obtain the values of the significant computational parameters. The first method uses a separate benchmark to obtain these parameters. We observed that the parameter values for local operations approximated by this method were too low, while the parameter values for remote operations were too high. We suspect that the accuracy of this method can be improved by moving towards linear functions of table size instead of the single values which are now used as plateaus. In the second method, an existing benchmark was adapted to perform different query mixes. This *in vivo* method produced better results. With the majority of the predicted results for both platforms being within a 16% margin of the actual measured result, we deemed this method to be accurate enough for our purposes of analyzing whether or not offloading computational load from the main server is beneficial, even though the accuracy of the predictions can be improved.

Using this model with the *in vivo* approximation method, the characteristics of offloading computational load of a main database server has been studied using a local cache on two reference architectures. When computational load is offloaded from the main server, all read actions are performed on the local cache. In this case, all write actions must be forwarded to the main DBMS. The model shows that, within this setup, the speedup going from a remote read to a local read is a very dominant parameter and that an overall speedup of a certain query mix can only be obtained when there are sufficient read actions present. Even though accuracy of the predictions can be improved, we argue that performance models, like those proposed in this chapter, are well-suited to support an analysis to determine whether offloading computational load is beneficial.

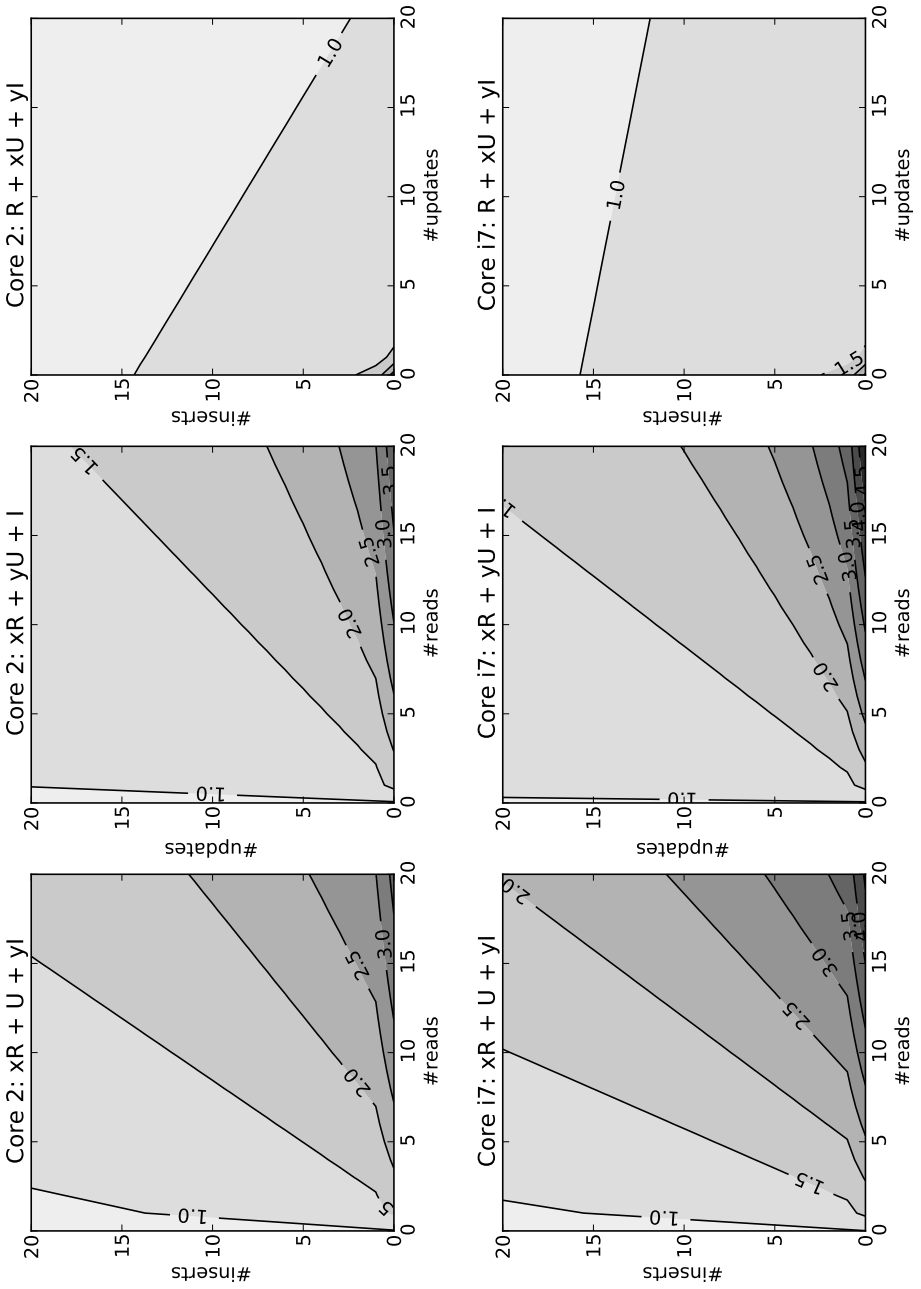


Figure 8.2: Contour plots showing the speedup of remote operation versus local operation and forwarding of updates for different query mixes and different platforms.

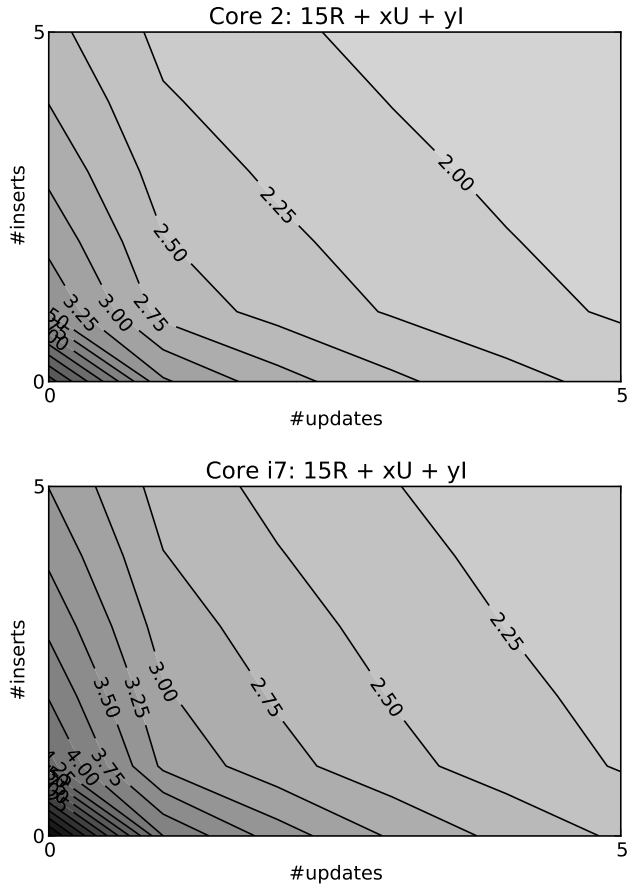


Figure 8.3: Contour plots showing speedup of remote operation versus local operation and forwarding of updates for different query mixes performing 15 reads.

PART II

Tuple-Based Optimization of Irregular Codes

CHAPTER 9

Transformations for Automatic Data Structure Reassembly

9.1 Introduction

The first part of this thesis discussed how the *forelem* framework arose from the unification of code optimization in seemingly distinct fields of programming: transactional (database) applications and other (imperative) applications, and how this framework unifies these distinct fields of programming by expressing queries in an intermediate representation as a series of tuple accesses governed by simple loop control. Subsequently, this intermediate representation is optimized by traditional optimizing compiler techniques, accomplishing results similar to query optimization. In Chapter 7 the *forelem* framework was used to perform vertical integration of database applications, where queries in a database application are replaced with code segments that evaluate these queries using direct access to a local data store. Subsequently, the application and data access codes are optimized together.

In the second part of this thesis, the foundation of the *forelem* framework will be generalized and the use of the *forelem* framework for different code optimization problems will be discussed. Because the *forelem* framework was initially envisioned for database applications, its main features rely on viewing data as being stored as (multi)sets of tuples. The access of data through a tuple space is thus the main characteristic of the *forelem* framework. As a consequence, we propose problems from different application domains to be (automatically) expressed in terms of tuples, which enables the *forelem* framework to be used for optimization of these problems. For example, sparse matrix computations are characterized by the fact that next to the values, the column index and row index play an essential role. It is this relation, which can be naturally expressed as a tuple.

Next to accessing data as tuples, the *forelem* framework allows the execution order of tuple computations (transactions) to be out of order. This feature together with the possibility of presenting data access without having to specify the exact

data storage enables the *forelem* framework to automatically generate storage formats. Because of this out of order execution, application of compiler optimizations has to be carefully handled. As standard compiler optimizations rely on data dependence analysis and loop-carried dependencies, and these loop-carried dependencies are non-existing in *forelem* loop nests, the conditions under which the transformations can be applied have to be reconsidered, as has been discussed in Chapter 3.

In this chapter, the *forelem* framework is extended to define compiler transformations that operate on three levels: the tuple level, the materialized loop index level and the concretized data access level. The *forelem* tuple level provides an elegant representation method for expressing different data access codes such as database queries and sparse matrix algebra. Within the materialized loop index level, index sets on the tuple space that specify access patterns are being represented as array accesses. This is done without specifying how the tuples or arrays are actually stored. By giving the compiler transformation framework access to this second level of data access, the compiler can address the order of data access while the order of execution is not specified. Finally, within the concretized data access level, loops are expressed using regular (integer) iteration bounds. At this level, standard compiler optimizations can be applied taking into account the different semantics for data dependencies.

This chapter is organized as follows: in Section 9.2 the *forelem* intermediate representation is reiterated and generalized such that it applies to irregular computations as well. Section 9.3 demonstrates how Sparse BLAS routines are expressed in the *forelem* intermediate representation. Section 9.4 introduces the orthogonalization transformation, that can be used to impose a certain order on the iteration of the data. This is a preparatory step to materialization, discussed in Section 9.5. In this section, the process of transforming a loop to the materialized loop index level is defined and several transformations applicable at the materialized loop index level are described. Section 9.6 describes a number of transformations that can be applied on materialized *forelem* loops, influencing the data storage format that is generated. Section 9.7 outlines how loops are converted to the concretized data access level. In Section 9.8, the results of initial experiments performed with an important kernel, sparse matrix times k vector multiplication, are presented. Section 9.9 concludes this chapter.

9.2 The Forelem Intermediate Representation

In this section, the basics of the *forelem* intermediate representation as introduced in Chapter 3 will be briefly reiterated and be generalized such that it applies to irregular computations as well. As has been discussed, the intermediate representation is centered around the *forelem* loop construct. Each *forelem* loop iterates over a specific array of structures. The subscripts of this array that are accessed are fetched from an "index set" that is associated with the array.

The arrays of structures that are iterated by *forelem* loops are modeled after database tables which are defined as multisets. The structure reflects the format of a database tuple. The intermediate representation operates at the tuple level, at

which is it not determined how the tuples are stored. For instance, the tuples are stored either row-wise or column-wise. In the latter case, a structure of arrays is iterated. In an array of structures `A` a tuple at index `i` is accessed with `A[i]` and a specific field *field1* in that tuple is accessed with `A[i].field1`.

An *index set* is a set containing subscripts $\mathtt{i} \in \mathbb{N}$ into an array. Since each array subscript is typically processed once per iteration of the array, these subscripts are stored in a regular set. Index sets are named after the array they refer to, prefixed with "p". For example, `pA` is the index set of all subscripts into an array `A`: $\forall s \in \mathtt{A} : \exists \mathtt{i} \in \mathtt{pA} : \mathtt{A}[\mathtt{i}] = s$. Random access of an index set by subscript is not possible, instead all accesses are done using the $\in$ operator.

The body of a *forelem* loop typically performs an action on the tuple subscripted by the current value of the loop iterator. When used in the context of database codes, the loop body often outputs tuples to a temporary or result set. Temporary sets are generally named $\mathscr{T}_1, \mathscr{T}_2, ..., \mathscr{T}_n$ and result sets $\mathscr{R}_1, \mathscr{R}_2, ..., \mathscr{R}_n$. In the context of, for example, sparse matrix codes a computation is typically performed also involving data from dense matrices or vectors. Results could be stored in a dense array.

Considering an array `A` with fields *field1* and *field2*, a *forelem* loop that iterates all entries of `A`, outputting the value of *field1* of each row, is written as follows:

```
forelem (i; i ∈ pA)
  ℛ = ℛ ∪ (A[i].field1)
```

Although the *forelem* loop appears to be very similar to a *foreach* loop that exists in many common programming languages, *forelem* loops distinguish themselves with the use of the index sets. Every *forelem* loop iterates a single array, using subscripts from an index set that is associated with that array. Note that, the order of the subscripts in the index set is undefined. The only thing that is defined is *which* subscripts are to be iterated, but not in which order. As such, *forelem* loops do not have explicit looping structures and the exact semantics of the iteration of an array are determined in the course of the optimization process. Index sets are the essence of *forelem* loop nests as they encapsulate iteration and simplify the loop control so that aggressive compiler optimizations can be successfully applied.

Using conditions on index sets it is possible to narrow down the range of the array that is iterated. For example, the index set denoted by `pA.field2[k]` contains only those subscripts into `A` for which *field2* has value `k`. This is expressed mathematically as follows:

$$\mathtt{pA.field2[k]} \equiv \{\mathtt{i} \mid \mathtt{i} \in \mathtt{pA} \wedge \mathtt{A[i].field2} = \mathtt{k}\}$$

So, to only iterate entries of `A` in which the value of *field2* is 10, the following *forelem* loop is used:

```
forelem (i; i ∈ pA.field2[10])
  ℛ = ℛ ∪ (A[i].field1)
```

Note, that `pA.field2[10]` is not expressed more explicitly as the exact execution of the loop will be determined during the optimization process. This index set

```
forelem (i; i ∈ pC)
{
  int sum = 0;
  forelem (j; j ∈ pA.row[C[i].index])
    forelem (k; k ∈ pB.index[A[j].col])
      sum += B[k].value * A[j].value;
  C[i].value = sum;
}
```

Figure 9.1: Matrix-Vector Multiplication with sparse vectors.

might be explicitly generated (at compile- or run-time), combined with other index sets, moved or eliminated. Alternatively, during the optimization process it may be decided to create a variant of array `A` only containing the tuples to be iterated.

More sophisticated index sets are possible, such as having conditions on multiple fields, in this case on *field1* and *field2*:

$$\texttt{pA}.(\texttt{field1},\texttt{field2})[(\texttt{k}_1,\texttt{k}_2)] \equiv \{\texttt{i} \mid \texttt{i} \in \texttt{pA} \wedge \texttt{A}[\texttt{i}].\texttt{field1} = \texttt{k}_1 \wedge \texttt{A}[\texttt{i}].\texttt{field2} = \texttt{k}_2\}$$

Instead of a constant value, the values $\texttt{k}_n$ can also be references to values from another array. To use such a reference, the array, subscript into the array and field name must be specified, e.g.: `A[i].field`. To select values $\texttt{field1} > 10$ an interval is used: $(10, \infty)$.

9.3 Expressing Sparse BLAS routines in Forelem

In this section, we will demonstrate how Sparse BLAS routines are expressed in the *forelem* intermediate representation. Sparse structures are considered to be sets of tuples. A sparse matrix is represented using tuples of the form `(row, column, value)`. Sparse vectors can be represented using `(index, value)`. When consider tables to only contain a single tuple for every unique `(row, column)` pair or `index`.

As a first routine, we consider the Matrix-Vector Multiplication $C = AB$. Figure 9.1 shows this multiplication where C and B are considered to be sparse vectors and are thus represented as tables[1]. Note the repeated use of index sets to define which tuples should be processed within an iteration. Figure 9.2 shows the *forelem* representation for the same operation, but with C and B are dense vectors.

Other BLAS routines can be similarly expressed. In Figure 9.3 an implementation of Triangular Solve $Tx = B$ using *forelem* loops to access a matrix T is

[1]For the interested reader, the SQL specification of this representation is: `select distinct A2.row, (select sum(B.value * A.value) from A, B where B.index = A.col and A.row = A2.row) from A A2;`

```
for (i = 1; i <= N; i++)
{
  int sum = 0;
  forelem (j; j ∈ pA.row[i])
    sum += B[A[j].col].value * A[j].value;
  C[i] = sum;
}
```

Figure 9.2: Matrix-Vector Multiplication with dense vectors.

```
for (i = N; i >= 1; i--)
{
  forelem (j; j ∈ pT.(col,row)[(i, i)])
    x[i] = b[i] / T[j].value
  forelem (j; j ∈ pT.col[i])
    b[T[j].row] = b[T[j].row] - T[j].value * x[T[j].col]
}
```

Figure 9.3: An implementation of Triangular Solve $Tx = b$ written in the *forelem* intermediate representation.

presented. As an additional example, Figure 9.4 shows an implementation of LU Factorization. Note that every loop over the same sparse matrix A defines a different set of matrix elements to be iterated.

9.4 Orthogonalization

In *forelem* loops, iteration of a table of tuples is controlled by the index set. No order is defined on the index set, which has as consequence that the iteration order of the table is undefined. In this section, the *orthogonalization* transformation is introduced, which makes it possible to impose a certain order in which the table is iterated. This is achieved by partitioning the accesses to the array based on the values of one or more table fields. The orthogonalization transformation is used to control the order in which data is accessed as a preparatory step to Materialization, which is discussed in the next section.

Let `A` be a table with `field1`, `field2`, ..., `fieldn`. Consider the loop:

```
forelem (i; i ∈ pA)
  ... A[i] ...
```

In this loop, the tuples of `A` can be iterated in any order. As an example, assume an iteration order is to be imposed on `A` such that tuples `A` are accessed in blocks with equal values for `field1`. The orthogonalization transformation is carried out to achieve this, resulting in the following loop nest:

```
for (i = 1; i <= N; i++)
{
  p = diag(i)
  forelem (j; j ∈ pA.(col,row)[(i, (i, ∞))])
  {
    A[j].value = A[j].value / p
    forelem (l; l ∈ pA.(row,col)[(i, (i, ∞))])
    {
      fillin = True
      forelem (k; k ∈ pA.(col,row)[(A[l].col, A[j].row)])
      {
        A[k].value = A[k].value - A[j].value * A[l].value
        fillin = False
      }
      if (fillin)
      A = A ∪ (A[k].row, A[k].col, - A[j].value * A[l].value)
    }
  }
}
```

Figure 9.4: An implementation of LU Factorization written in the *forelem* intermediate representation.

```
forelem (ii; ii ∈ A.field1)
  forelem (i; i ∈ pA.field1[ii])
    ... A[i] ...
```

`A.field1` in the outer loop denotes all possible values of `field1` that occur in `A`. So, the iteration space of the outer loop consists out of every value of `field1` in `A`.

The original loop iterates all tuples of `A`. The transformed loop nest will for every value of `field1`, iterate all tuples of `A` for which `field1` equals this value. As a result, the transformed loop also iterates all tuples of `A`. Application of the orthogonalization transformation is not limited to a single field. An example of orthogonalization on two fields is:

```
forelem (ii; ii ∈ A.field1)
  forelem (jj; jj ∈ A.field2)
    forelem (i; i ∈ pA.(field1,field2)[(ii,jj)])
      ... A[i] ...
```

The outer loops that are introduced by the orthogonalization transformation iterate all values of a given table field. If it is possible to express this range of values as a subset of the natural numbers, i.e. `A.field1` $\subseteq \mathbb{N}$, the *encapsulation* transformation can be applied, which replaces the loop over all table field values with a loop over a subset of the natural numbers. With the encapsulation transformation, a loop

forelem (ii; ii ∈ A.field1)

where A.field1 $= \{1, 2, 6, 7, 8, 10\}$, is replaced with:

forelem (ii; ii ∈ $\mathbb{N}_{10}$)

with $\mathbb{N}_{10} = [1..10]$. In the encapsulated loop, the values $3, 4, 5, 9$ will be iterated, but note that no tuple will exist where field1 equals any of these values. As a result, the inner loop is not executed for these values, maintaining the iteration space of the original loop.

9.5 Materialization

In this section, the materialization transformation is described, which materializes the tuples iterated by a *forelem* loop using the accompanying index set to an array in which the data is represented in consecutive order and is accessed with integer subscripts. Although this can be seen as a simple normalization operation, it is an important enabling step that allows the compiler to address and modify the order of data access to these arrays. In fact, by materialization the execution order of an inner loop is fixed. (In the case of nested loops, orthogonalization fixes the order of the outermost loop). After two forms of materialization have been introduced, a number of transformations targeting the order in which data access takes place will be described.

A distinction is made between loop-independent and loop-dependent materialization. In loop-independent materialization, conditions in the index set of the loop to be materialized are not dependent on one of the outer loops. Materialization will result in a one-dimensional array. In loop-dependent materialization, the resulting array will get an additional dimension for each dependent loop. Both cases of materialization will now be discussed in turn.

9.5.1 Loop Independent Materialization

We first consider loop-independent materialization. The following loop iterates all tuples of A whose field equals a value X:

```
forelem (i; i ∈ pA.field[X])
  ... A[i] ...
```

To be able to determine which tuples of A to access, the index set is used. This is, in fact, a indirection level. This indirection can be removed by materializing the index into the tuple space as an array PA which only contains the entries of A that should be visited by this loop. This results in:

```
forelem (i; i ∈ ℕ*)
  ...  PA[i]  ...
```

with $\mathbb{N}* = [1, |\texttt{PA}|]$. The array PA only contains elements from A for which the condition A[i].field == X holds. The compiler is now enabled to address the order

in which the data in PA is accessed, while the execution order of the loop is not specified. For example, using the transformations that can be applied on the materialization form, which are described below, the compiler can determine to put entries in PA in a specific order. The loop control is selected at the concretization stage, where the compiler can ensure the loop control for the loop will iterate the items of PA consecutively. For the general definition of loop-independent materialization, consider a loop iterating a sparse structure A:

```
forelem (i; i ∈ pA)
  ...  A[i]  ...
```

which is transformed to:

```
forelem (i; i ∈ ℕ*)
  ...  PA[i]  ...
```

with $\mathbb{N}* = [0, |\texttt{PA}|)$. This transformation materializes the sparse structure A to an one-dimensional array PA.

The transformation can also be applied if the loop to be materialized is nested in another *forelem* loop and the posed condition in the index set of the loop to be materialized is *independent* of the outer loop. Consider, for example, where the outer loop could be the result of the application of the encapsulation transformation:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ pA.field[X])
    ... A[j] ... B[i] ...
```

Materialization of the inner loop will enable the compiler to address the order of data access of A together with the other array or tuple space references. Materialization of the inner loop proceeds as explained above and the outer loop is untouched:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ*)
    ... PA[j] ... B[i] ...
```

with $\mathbb{N}* = [1, |\texttt{PA}|]$ and PA only containing items that satisfy the condition.

9.5.2 Loop Dependent Materialization

If a loop to be materialized is contained in a loop nest and the conditions of its index set have a dependency on another loop, then the above described loop-independent materialization cannot be applied. Instead, loop-dependent materialization must be used, which is described in this section. Because loop-dependent materialization will result in higher-dimensional arrays, this results in more opportunities for the compiler to address and modify the order of data access to these arrays. In general, a loop-dependent materialization has the form:

```
forelem (i; i ∈ ℕ_o)
  ...
    forelem (n; n ∈ ℕ_t)
      forelem (p; p ∈ pA.(field_i, ...,field_n)[(i,...,n)])
         ... A[p] ...
```

The index set iterated in the inner loop has a dependency on one or more of the outer loops. The iteration of `A` is materialized to an iteration of a multi-dimensional array `PA`, in which each loop-dependent condition is represented as an additional dimension in `PA`. The array `PA` only contains these items that are iterated by the original index set on `A`:

```
forelem (i; i ∈ ℕ_o)
  ...
    forelem (n; n ∈ ℕ_t)
      forelem (p; p ∈ ℕ∗)
        ... PA[i]...[n][p] ...
```

with $\mathbb{N}* = [0, |\texttt{PA[i]...[n]}|)$. After this transformation, `PA` only contains entries that satisfy the conditions of the original index set. The dimensions of the materialized array correspond with the original conditions and thus with the loops on which the condition depended. Loop transformations, such as Loop Interchange, will thus have an effect on the order in which the data of `PA` is accessed. By taking this into account, the compiler can determine an efficient order in which the store the elements of `PA`, which has at this point not been set in stone.

To illustrate the loop-dependent materialization, consider a simple nested loop:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ pA.row[i])
    ... A[j] ...
```

The index set of the inner loop, `pA.row[i]` is dependent on iterator `i` of the outer loop. As a consequence, the array `PA` will obtain a dimension for this iterator `i`. The result of the materialization transformation is as follows:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ∗)
    ... PA[i][j] ...
```

with $\mathbb{N}* = [0, |\texttt{PA[i]}|)$. Because `i` was determining which row of `A` was iterated, in the transformed loop `i` still controls the order in which the rows of the original matrix `A` are accessed in the materialization `PA`.

In case the index set has dependencies on two loops, a three-dimensional array is generated. Naturally, this has more degrees of freedom for optimization than the two-dimensional materialization. The application of the transformation is similar as in doubly-nested loops. In this example, the index set has dependencies on two different outer loops:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ_m)
    forelem (k; k ∈ pA.(row,col)[(i,j)])
      A[k].value = ...
```

This results in a three-dimensional array PA:

```
forelem (i; i ∈ ℕ_n)
  forelem (j; j ∈ ℕ_m)
    forelem (k; k ∈ ℕ*)
      PA[i][j][k].value = ...
```

with $\mathbb{N}* = [0, |\texttt{PA[i][j]}|)$.

9.5.3 Materialization Combined with Other Transformations

We will now demonstrate how the combination of materialization with other transformations within the *forelem* framework leads to the automatic generation of different data storage formats for a particular problem. In this section, Matrix-Vector Multiplication is considered with a sparse matrix A and dense vectors B and C:

```
forelem (i; i ∈ ℕ_m)
  C[i] = 0;

forelem (i; i ∈ ℕ_m)
  forelem (j; j ∈ pA.row[i])
    C[i] += B[A[j].col] * A[j].value;
```

In the remainder of this example we will focus on the second loop and consider that the first loop initializing C is left untouched. On the inner loop of this second loop nest, the materialization transformation will be applied. Because the argument to the index set of the inner loop depends on the outer loop, loop-dependent materialization will be performed. This will result in a two-dimensional array PA:

```
forelem (i; i ∈ ℕ_m)
  forelem (j; j ∈ ℕ*)
    C[i] += B[PA[i][j].col] * PA[i][j].value;
```

Note that the arrays PA[i] will for every i only contain these elements of A that satisfied A[j].row == i. As a next step, the loops are interchanged using the Loop Interchange transformation. This is possible because no loop-carried dependencies are present as the *forelem* loops do not guarantee a specific iteration order:

```
forelem (j; j ∈ ℕ*)
  forelem (i; i ∈ ℕ_m)
    C[i] += B[PA[i][j].col] * PA[i][j].value;
```

The iterator j still controls which entry within the current row (indicated by i) is visited. These entries may not necessarily have the same column index, as entries

which are zero are not present in PA. So, this loop nest will for each column number (outer loop) iterate all rows. This is the essence of the jagged diagonal storage format, which in consecutive rows stores all first nonzero column entries of all rows, all second nonzero column entries of all rows, and so on. The corresponding column indices are stored in a separate array. Important is that this particular storage format has been deduced without any predefinition of this format in the framework. As will be described in Section 9.7, different variants of this jagged diagonal storage format can be concretized.

9.6 Transformations on the Materialized Form

After a *forelem* loop has been put in a materialized form, the data to be processed has been put in an array in consecutive order and is accessed with integer subscripts. At this stage, the compiler can modify the exact order of data access to these arrays and how this data is stored. In this section a number of transformations are described that affect the storage of the data processed by a loop nest.

9.6.1 Horizontal Iteration Space Reduction

The aim of Horizontal Iteration Space Reduction is to reduce unused fields from a table's schema. In fact, it is possible to perform this transformation before the materialization stage.

Formally, the transformation is defined as follows. Let T be a table with fields field1, field2, field3, field4, and C a list of condition fields $C \subset$ (field1 field2) and V a list of values. Consider the loop nest:

```
forelem (k; k ∈ pT.C[V])
  ℛ = ℛ ∪ T[k].field1 + T[k].field2
```

We define a new table $T' \subseteq$ T with fields field1, field2 and replace the use of T with T' in the loop.

9.6.2 Structure splitting

Before materialization tables are represented as multisets of tuples, accessible with integer subscripts. By default, the array that is the result of the materialization operation is an array of tuples or structures. In some cases, it is more efficient to use a structure of arrays, i.e. the structures are split [94, 26]. Within the *forelem* framework this is defined as the *structure splitting* transformation. Consider the materialized loop nest:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA[i][k].value ...
```

Structure splitting will modify the data storage of the array and convert the data accesses in the loop to:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA.value[i][k] ...
```

9.6.3 $\mathbb{N}*$ materialization

Materialized loops use the $\mathbb{N}*$ index set as the set of integer subscripts to access the materialized array. How exactly these integer subscripts are stored is initially encapsulated within $\mathbb{N}*$ and can be made explicit using $\mathbb{N}*$ materialization. Consider the following loop, the result of a materialization to `PA`:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ ℕ*)
    ... PA[i][k] ...
```

As a prerequisite for the final code generation stage, $\mathbb{N}*$ must be made explicit. This can be achieved by converting $\mathbb{N}*$ to a set `PA_len`. There are different means in which this set can be defined. The first is to define the set as follows:

```
PA_len[q] = max(len(PA[q]))
```

in which case all `PA_len[q]` values are the same and a single set containing integers up to the maximum value can be stored for this loop nest. Padding is inserted in the array `PA` for the values `PA[i][k]` with `k >= PA_len[i]`. The second way to create this array is to avoid inserting padding in `PA`. In this case `PA_len[q] = len(PA[i])`.

Regardless of which implementation is chosen, the resulting loop after $\mathbb{N}*$ materialization is:

```
forelem (i; i ∈ ℕ_m)
  forelem (k; k ∈ PA_len[i])
    ... PA[i][k] ...
```

Note that in this loop the iteration order is still undefined. Only $\mathbb{N}* = [0, \mathbb{N}*)$ has been replaced with $\texttt{PA\_len}[\texttt{i}] = [0, \texttt{PA\_len}[\texttt{i}])$. In a subsequent concretization step the iteration order will be determined. For example, the loop:

```
forelem (k; k ∈ PA_len[i])
```

is concretized to:

```
for (k = 0; k < PA_len[i]; k++)
```

9.6.4 $\mathbb{N}*$ sorting

In case of loop-dependent materialization, $\mathbb{N}*$ encapsulates the sets of integer subscripts used for iteration of the inner loop. These sets are ordered irrespective of their cardinality. If the loop is to be parallelized, it is beneficial if the work is divided into blocks with evenly sized values for `PA_len` (after $\mathbb{N}*$ materialization). One way to achieve this is by imposing an order on the iteration of $\mathbb{N}*$.

The aim of $\mathbb{N}*$ sorting is to find an order of the iterator values `i` such that the value of $\mathbb{N}*$ decreases with subsequent iterations of the outer loop on `i`:

```
forelem (i; i ∈ ℕm)
  forelem (k; k ∈ ℕ∗)
    ... PA[i][k] ...
```

Consider that $\mathbb{N}* = [0, \texttt{len}(\texttt{PA}[\texttt{i}]))$. The goal is to iterate through $\mathbb{N}_m$, such that `len(PA[i])` decreases. Let $\texttt{perm}(\mathbb{N}_m)$ store the permutation of $\mathbb{N}_m$ for which this holds. Then, the loop is transformed to:

```
forelem (i; i ∈ perm(ℕm))
  forelem (k; k ∈ ℕ∗)
    ... PA[i][k] ...
```

Note that this will affect the order of the data `PA`, which will be put in the corresponding sorted order at the concretization stage.

9.6.5 Dimensionality Reduction

Loop-dependent materialization results in a multi-dimensional array by default. If this array is concretized as a multi-dimensional array, padding may have to be inserted for the uneven lengths of the rows. It is possible to avoid the introduction of this padding by storing the rows back to back. This reduces the dimensionality of the materialized array. Consider the loop nest:

```
forelem (i; i ∈ ℕm)
  for (k = 0; k < PA_len[i]; k++)
    ... PA[i][k] ...
```

to reduce the dimensionality of the materialized array `PA` by one, this is transformed into:

```
forelem (i; i ∈ ℕm)
  for (k = PA_ptr[i]; k < PA_ptr[i+1]; k++)
    ... PA[k] ...
```

Based on the `PA_len` array, a new `PA_ptr` array is introduced, which keeps track of the start and end of each row in `PA`. Note that the order of the iteration domain $[\texttt{PA\_ptr}[\texttt{i}], \texttt{PA\_ptr}[\texttt{i} + 1])$ does not have to be defined and could be in any order.

9.7 Concretization

Concretization is a simple one-to-one mapping from a given materialized loop to a C *for* loop that can be compiled by a regular C compiler. So, a *forelem* loop iterating a subset of integers is transformed into a regular *for* loop. This transformation selects a specific iteration order for the subset of integers. Essentially, at this point the data storage format is generated that has been chosen by the optimization process. Using the different transformations that can applied on a materialized loop, described in the preceding section, many different storage formats can be generated for a single loop nest.

To describe the basic concretization transformation, consider the following loop as an example, which is the result of a materialization transformation:

```
forelem (i; i ∈ ℕ*)
  ... PC[i] ...
```

As a first step, $\mathbb{N}*$ materialization is applied, resulting in:

```
forelem (i; i ∈ PA_len)
  ... PC[i] ...
```

then the loop can be subsequently concretized to:

```
for (i = 0; i < PA_len; i++)
  ... PC[i] ...
```

At this point, a data storage format has been chosen by the optimization process. For this particular, single-dimensional, case, storage as an array of consecutive values is the most likely candidate. Note that, this storage format is the result of just merely a straightforward mapping of the materialized index set into a (multi)dimensional data structure. This cannot be compared to substituting coordinate storage by jagged diagonal storage, or the immediate selection of such a pre-defined format.

To better illustrate the possibilities within the concretization process, we will continue the Sparse Matrix-Vector Multiplication example from Section 9.5.3:

```
forelem (j; j ∈ ℕ*)
  forelem (i; i ∈ ℕ_m)
    C[i] += B[PA[i][j].col] * PA[i][j].value;
```

In this example the data access to `PA` is to be concretized. As a first step, the loop with iterator variable `i` is concretized:

```
forelem (j; j ∈ ℕ*)
  for (i = 1; i <= m; i++)
    C[i] += B[PA[i][j].col] * PA[i][j].value;
```

Given that iterator variable `j` indicates which column number to process, the concretized inner loop will now iterate all rows to process in consecutive order. The outer loop can be concretized to iterate the column numbers in consecutive order. To do this, first $\mathbb{N}*$ materialization is applied. If $\mathbb{N}*$ is converted to a set `PA_len` such that `PA_len[q] = max(len(PA[q]))`, all values in the set are the same, so a further conversion is possible to a single constant value, say `k`. The outer loop can then be concretized to result in:

```
for (j = 1; j <= k; j++)
  for (i = 1; i <= m; i++)
    C[i] += B[PA[i][j].col] * PA[i][j].value;
```

These steps have led to a certain storage scheme for `PA`. This storage scheme consists of a two dimensional array which has a row for each column number n, containing all nonzero column entries at position n in the different rows. Unused entries are padded with zero, so that every row has the same length. This enables

that a generic two dimensional array can be used as storage scheme. If code is generated for the C language, which uses row-major order for array storage, then the rows containing the column values and indices must be stored one after the other. The resulting array should be accessed with `PA[column][row]`, which is in fact different from the order of the subscripts in the current *forelem* representation. So, as a result the following C code will be produced for this storage format:

```
for (j = 1; j <= k; j++)
  for (i = 1; i <= m; i++)
    C[i] += B[PA[j][i].col] * PA[j][i].value;
```

where `k` will be a constant indicating the maximum number of non zero columns in a row in the resulting array `PA`.

Different transformations could have been applied in between the materialization and concretization steps to influence the data storage format that is generated in the end. For example, the `row` field that is part of the original tuples is not used in the code fragment. Using Horizontal Iteration Space Reduction, such unused fields are eliminated. It is also possible to store the `col` and `value` fields in separate arrays. To accomplish this, the structure splitting transformation must be performed before concretization. The concretized result in C code will be:

```
for (j = 1; j <= k; j++)
  for (i = 1; i <= m; i++)
    C[i] += B[PAcol[j][i]] * PAvalue[j][i];
```

Note that this generated storage scheme is described in the literature as simplified Jagged Diagonal Storage, or ITPACK storage [12]. So, through the described transformations, orthogonalization, materialization and concretization, many different loops and accompanying data storage formats can be generated that achieve the same result. Figure 9.5 illustrates how such data formats are generated, starting from an unordered set of tuples. Established data storage formats, such as ITPACK and Jagged Diagonal Storage format [12], simply follow from the application of the transformations described in this chapter. For example, the transformation sequence drawn in black in this figure is the sequence leading to the ITPACK storage. The $\mathbb{N}*$ materialization step is considered to be part of the concretization step in this figure. Alternatively, when the structure splitting transformation in the figure is followed by dimensionality reduction, Compressed Row Storage (CSR) format is generated. Similarly, a transformation sequence that continues from orthogonalization on column can result in Compressed Column Storage (CCS) format.

Let us consider a different application of the transformations. If the alternative form of $\mathbb{N}*$ materialization is applied, a set `PA_len` is generated such that no zeros have to be inserted into the data structure as padding. Through the application of dimensionality reduction, the rows stored in memory, that thus contain column entries, will be stored back to back in a vector. As a consequence, an additional data structure is added to record the start of each row. When $\mathbb{N}*$ sorting is applied, rows with similar number of entries are placed close to each other, for example by reordering the rows of the matrix by sorting on the number of non-zeros per

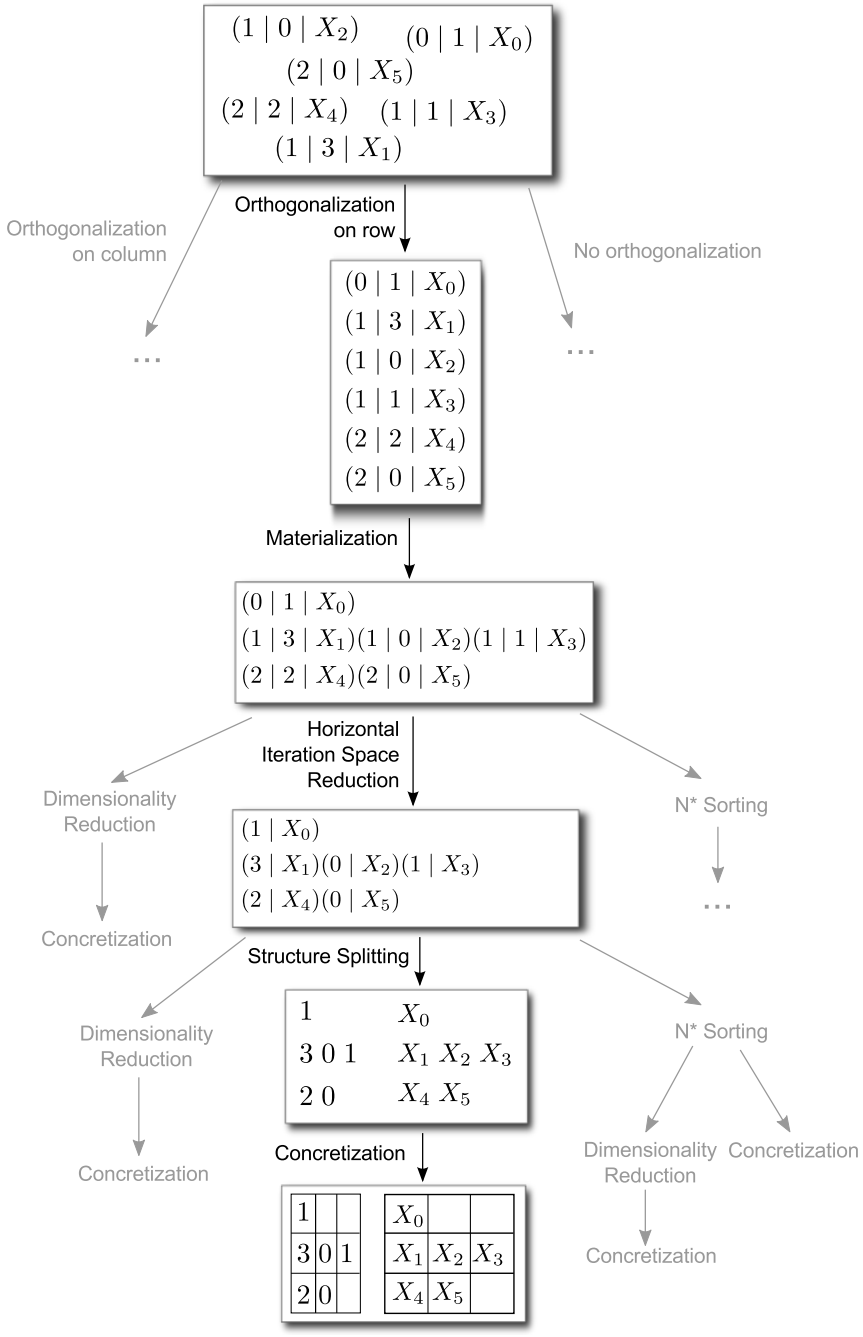


Figure 9.5: An illustration of the application of orthogonalization, materialization and concretization on sparse matrix tuples in (`row` | `col` | `value`) format. The result of this concretization is commonly known as the ITPACK format (assuming the arrays are stored in column-major order). The arrows displayed in gray depict a non-exhaustive set of other possibilities.

row. This can be helpful as a form of load balancing in the case of parallelization. Although this transformation changes the order in which the rows are processed, this does not introduce a problem because before the concretization the iteration of the rows is specified as a *forelem* loop which does not impose a particular execution order. Note that this concretization leads to the Jagged Diagonal Storage format described in the literature [12]. This storage format follows from the application of the generic transformations, contrary to be devised by hand which was the only way to arrive this storage format up till now.

9.8 Initial Experimental Results

In this section, initial experimental results are presented of the performance of codes and data storage formats optimized with the *forelem* framework. It is demonstrated that these optimized codes are comparable in performance to hand-optimized sparse routines using the CUDA framework. As an example, the sparse Matrix-Vector Multiplication will be considered, with a sparse matrix A and dense vectors B and C. From the materialization of this loop that has been described in the previous section, three different concretizations have been generated for which CUDA code has been generated[2]:

- *Simple JDS*: in this case, every row contains the nonzero column entries of a single row in consecutive order. This is thus the result of a concretization if **no** Loop Interchange was performed during the materialization.
- *Simple JDS 2*: in this case, every row contains the nonzero column entries at position n of every row. This is the format generated in that previous section that is similar to ITPACK.
- *JDS*: Jagged Diagonal Storage format, as described in the previous section.

These generated implementations have been compared with Matrix-Vector Multiplication as implemented in the CUSP [14] library for different storage formats. The CUSP library provides several routines for performing sparse linear algebra on CUDA, which have been optimized for several, pre-defined, storage formats. The formats implemented in CUSP that we have benchmarked are: Coordinate format (COO), Compressed Sparse Row format (CSR) and a Hybrid (HYB) format. The Hybrid format employs a combination of the ELL format and COO format and is described in detail in [14].

For the benchmark, 15 matrices have been selected from The University of Florida Sparse Matrix Collection [28], also taking into account previous studies on sparse matrix times vector multiplication see [14, 98]. The selected matrices are listed in Table 9.1 and represent different problem classes.

The experiments have been performed on a workstation with a GeForce GTX 480 CPU with 1535MB of RAM. The multiplication operation has been repeated 1000 times. Figure 9.6 reports the execution time in milliseconds of 1000 Matrix-Vector Multiplications for the different matrices and implementations.

[2]In the next chapter, the search space of all possible concretizations will be explored and characterized

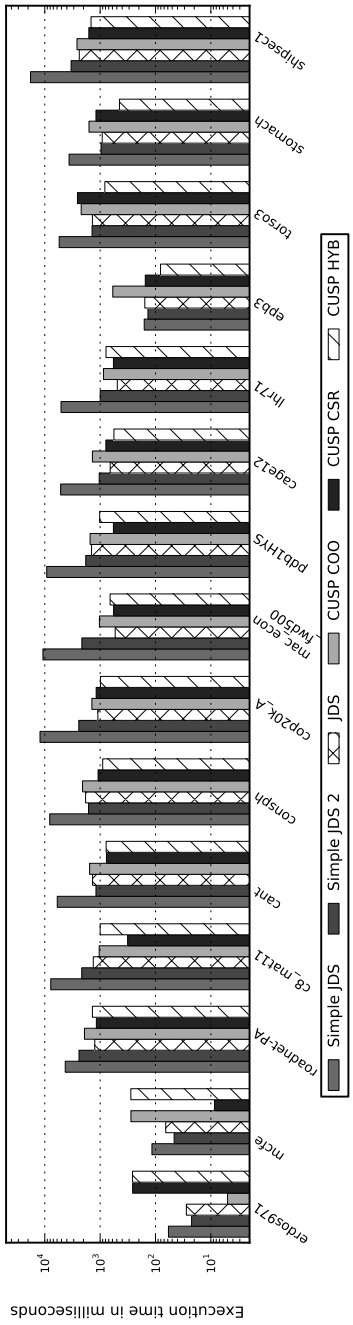


Figure 9.6: Execution time in milliseconds of 1000 Matrix-Vector Multiplications for different matrices and implementations. Simple JDS, Simple JDS 2 and JDS are implementations generated with the *forelem* framework. CUSP COO, CUSP CSR and CUSP HYB are CUSP implementations using different pre-defined storage formats.

Matrix name	Dimensions	Nonzeros
erdos971	472 x 472	2628
mcfe	765 x 765	24382
roadnet-PA	1090920 x 1090920	3083796
c8_mat11	4562 x 5761	2462970
cant	62451 x 62451	4007383
consph	83334 x 83334	6010480
cop20k_A	121192 x 121192	2624331
mac_econ_fwd500	206500 x 206500	1273389
pdb1HYS	36417 x 36417	4344765
cage12	130228 x 130228	2032536
lhr71	70304 x 70304	1494006
epb3	84617 x 84617	463625
torso3	259156 x 259156	4429042
stomach	213360 x 213360	3021648
shipsec1	140874 x 140874	3568176

Table 9.1: Details of the matrices that have been used in the benchmark.

The results indicate that the performance of the *forelem* generated kernels using automatically generated storage formats is in many cases comparable to the hand-optimized CUSP routines. Due to the varying characteristics of the different matrices, there is no all-round best storage format. For the majority of the matrices however, the Simple JDS 2 and JDS implementations clearly perform better than the Simple JDS implementation.

9.9 Conclusions

In this chapter, we have described the extension of the existing *forelem* framework with materialization techniques. These techniques enable the compiler to address the order of data access in a loop, while the order of execution of the loop is not specified. Through different applications of the materialization and concretization transformations, different data storage formats can be automatically generated.

As an application of these techniques, we demonstrated how Sparse BLAS routines can be expressed in the *forelem* representation and how the *forelem* framework automatically generates data storage formats and implementations of the data access codes. Initial experimental results demonstrated the effectiveness of this approach. Three *forelem*-generated CUDA implementations of sparse matrix-vector multiplication were compared to three hand-optimized CUDA implementations using three different, pre-defined, storage formats. The results show that the implementations generated with the *forelem* framework show performance that is comparable to hand-optimized code.

The Jagged Diagonal, ITPACK and ELLPACK data storage formats have been exemplified in quite some papers in the past. The main reason is that sparse matrix times vector multiplication is the main kernel for many large-scale simulations. In this chapter, we have seen that these storage schemes naturally arise from a basic

compiler transformation as Loop Interchange combined with index set materialization and therefore can be automatically derived.

CHAPTER 10

Search Space Characterization

10.1 Introduction

Sparse matrix computations are an important class of compute intensive codes and are extensively used. Not surprisingly, many techniques have been developed to optimize sparse matrix computations. An important technique is the selection of a smart data structure for storing the sparse matrix corresponding to the computation to be carried out. However, wrapping the sparse matrix data in a specific data structure obscures the compiler optimization process and thus forms a major obstacle for further effective optimization and code generation.

As a consequence, many HPC applications rely on the use of sparse algebra run-time libraries to provide efficient, hand-optimized, implementations of common sparse matrix operations. However, given the complexity of today's CPU and GPU architectures, predetermined implementations of sparse algebra routines cannot get maximum performance out of the architecture. This is also the case for novel implementations based on expression templates to optimize performance, such as Blaze [43].

Besides the continuing advances in the target architectures, the parametrized nature of supplied sparse algebra routines and the fact that these routines are implemented for a limited number of data layouts also inhibit maximum performance from being achieved. For example, sparse matrix times matrix multiplication can be performed for a right-hand matrix with different numbers of columns k. Commonly, sparse algebra libraries provide a single routine for this multiplication that is parametrized for k. However, there is no single implementation of this computation that is a best fit for all possible parameters. Similarly, no sparse storage format exists that is optimal for different computations, different parameters for a computation or different sparse matrices. In short, predefined implementations based on predefined data layouts can never achieve optimal performance. Furthermore, the way these routines are implemented, by abstracting, or obscuring, the matrix data into a specific data structure, hampers optimizing compilers from producing more efficient codes.

Therefore, instead of maintaining predetermined implementations of sparse algebra routines in run-time libraries, we argue for a transition towards automatic instantiation of sparse algebra routines. In this chapter, we introduce computation-driven reassembly of sparse data structures, which is a key component of this automatic routine instantiation process. By combining sparse data structure reassembly with the code instantiation process, the construction of optimal data structures is made an integral part of the code optimization process. With computation-driven reassembly, an optimal sparse matrix storage format is derived from the actual sparse matrix computation, contrary to selecting a predefined storage format as is done in existing methods. In this approach, code and data layout are optimized hand in hand.

The automatic instantiation of sparse algebra routines and data layout reassembly is implemented as a series of code transformations in the *forelem* framework [83]. Within the *forelem* framework, all data is accessed through a tuple space. Data to be processed is specified as (multi)sets of tuples. In this case, the tuples arise from "disassembling" the original sparse matrix structure. The computation is expressed in terms of loops processing the tuples. Different transformations are implemented in the framework, ranging from standard compiler optimizations, such as Loop Interchange [4], Loop Fusion [52], Scalar Expansion and Def-Use analysis [2, 50], to transformations that address the order in which tuples are executed and stored. Among these latter transformations is Materialization, which is an important, enabling, transformation for computation-driven data structure reassembly.

Because the optimization process is made responsible for automatic generation of routines, the search space of this process is significantly enlarged. We demonstrate that more than 130 principal forms[1] of sparse matrix times k vector(s) and sparse matrix times matrix multiplication can be instantiated, with over 25 different reassemblies of the original sparse matrix data structure. *So, essentially, 25 different data structures are being generated.* This exemplifies the strength of our approach when compared to sparse algebra libraries, that on average implement 4, or less, pre-defined sparse data storage formats. When combined with parametric compiler optimizations, such as loop unrolling and loop blocking, the search space is enlarged by two to three orders of magnitude. A characterization of the possible instantiation search space shows that there are many different optimal instantiations. As a consequence, it is very hard, if not impossible, to predict which instantiation would be optimal for a given matrix, computation, computation parameters and architecture instance.

In addition to the search through automatically instantiated sparse algebra routines, a search through the parametric optimization search space, set up by transformations such as loop blocking and loop unrolling, can be done. This parametrized search space is complementary to the search space set up by the transformations leading up to differently structured computations and storage formats. We demonstrate that it is important to consider this second search space, as, for a single matrix, an instantiation that is optimal for unroll level a, is not necessarily the optimal instantiation for an optimal unroll level b. We show that

[1] Actually, 200 principal forms were generated, but for the experimentation 70 of these were deleted because they were too inefficient and therefore caused unacceptably long experimentation times.

with an exhaustive search through these combined search spaces, an automatically instantiated routine can be found that mostly outperforms the implementations from existing sparse algebra libraries or at least is equivalent in performance.

This chapter is organized as follows. Section 10.2 describes how different routines are instantiated and how data structure reassembly is carried out. In Section 10.3, an initial exploration is done of the search space of the different instantiations of sparse matrix times k vector(s) multiplication. Section 10.4 quantifies the irregularity of this search space using rank correlations. Section 10.5 discusses the irregularity observed in two other sparse matrix kernels. Section 10.6 presents the results of the comparison of our approach to a number of existing sparse algebra libraries. Section 10.7 presents our conclusions and plans for future work.

10.2 Reassembling Data Structures

This section describes the process and techniques that are used to instantiate efficient sparse algebra routines and reassemble the original sparse matrix data storage automatically. Using these techniques, many different forms of a sparse algebra routine can be generated, along with different reassemblies of the original sparse data storage.

As these techniques are implemented in the *forelem* framework [83], we first briefly introduce this framework. The principal syntactic construct in the *forelem* framework is the *forelem* loop. A *forelem* loop iterates (a subset of) a multiset of tuples and performs an operation on these tuples. As an example, consider a multiset `T` containing tuples with fields `field1` and `field2`: `(field1, field2)`. Then, the following loop sums the values of `field1` of tuples of which `field2` equals the value `9`:

```
sum = 0;
forelem (i; i ∈ pT.field2[9])
  sum += T[i].field1;
```

Iteration of the *forelem* loop is controlled with the "index set" `pT.field2[9]`, which in this case contains all subscripts into `T` for tuples of which `field2` equals `9`. The index set specifies which tuples will be visited, but does not specify the order in which these tuples are visited, which is undefined.

For the sparse algebra kernels to be expressed as a *forelem* loop, the sparse data structures are stored as sets of tuples and the dense data structures remain dense. A sparse matrix A can be represented as a set of tuples of the form `(row, column, value)`. Based on this, we can express a loop computing the sparse matrix vector product $c = Ab$ as follows:

```
forelem (m; m ∈ pA)
  C[A[m].row] = C[A[m].row] + B[A[m].col] * A[m].value
```

The expression of the computation that is performed by a sparse algebra routine in terms of tuples is the first step in the code instantiation process. The tuples arise from "disassembling" the original sparse matrix data structure. All non-zero

matrix elements are extracted from this original structure and are represented as tuples, one tuple per non-zero element.

On this initial specification of the sparse computation, various transformations can be applied that may modify the order in which the tuples are accessed and thus may influence how the tuples are reorganized. Since in *forelem* loops no explicit order is imposed on the iteration, tuples may be visited in any order. However, through a transformation known as Orthogonalization [83], a certain order can be imposed on the iteration based on the value of the fields of the tuples. A possible result of Orthogonalization is the following loop nest, which processes the above computation on a row-by-row basis:

```
for (i; i ∈ ℕn)
  forelem (m; m ∈ pA.row[i])
    C[i] = C[i] + B[A[m].col] * A[m].value
```

where $\mathbb{N}_n$ is the number of rows in matrix `A` and the index set `pA.row[i]` makes the *forelem* loop only iterate tuples in `A` with the value of field `row` equal to `i`.

To move towards a concrete implementation of this computation from this point, the Materialization transformation performs an important enabling role. The purpose of Materialization is to materialize the tuples iterated by a *forelem* loop using the accompanying index set to an array in which the data is represented in consecutive order and is accessed with integer subscripts. So, at this point tuples are physically reorganized into a particular order, based on the computation. Although this transformation can be seen as a simple normalization operation, it is an important enabling step that allows the compiler to address and modify the order of data access to these arrays. In fact, by materialization the execution order of an inner loop is fixed.

Note that these two transformations cause the original sparse data structure to be reassembled on a row-by-row basis. A compiler performing these transformations can thus exert control on the order in which data is stored. Up till now, optimizing compilers could not exert this amount of control on data storage order. Techniques have been described in the literature that do modify data structures, such as structure splitting [23], array regrouping [103] and field reordering [23], but these techniques are limited to rearranging data stored in arrays of structures in order to improve cache usage. Through Materialization much more invasive transformations of the data structure are enabled, such as translation from an unordered set of tuples to separate sequences of column indices and values stored in the order of ascending row number. See also the previous chapter.

Finally, after all transformations have been carried out, an implementation in C code is generated from the tuple program *and* a reassembled copy of the sparse data structure is instantiated based on the organization of tuples selected by the optimization process.

10.3 The Transformation Search Space

Using the approach described in the previous section, many different instantiations of the same sparse matrix routine can be generated. Different instantiations

can be distinguished because different transformations were performed before and/or after Materialization or transformations were performed in a different order. This section presents the results of the initial exploration of the search space of instantiations of sparse matrix times k vectors multiplication.

The full transformation tree of sparse matrix times k vector multiplication is shown in Figure 10.1. The starting point of the transformation space is labeled with *1* and shown in the center of the figure. This is the minimal representation of the computation as a *forelem* loop. From this point, there are several different branches of transformations as shown in the picture, resulting in many different variants, or principal forms. Whenever the label of a node is prefixed with "tmp", the node represents a stage for which no executable is generated. In all other cases, the executables (variants) are labeled from 1 to 130. Next to these 130 different implementations, also 25 different data structures are generated [2], ranging from simple coordinate storage to compressed row or column schemes, with or without zero-padded rows or columns, and jagged diagonal like schemes wherein the rows of the matrix have been permuted or not. For all these data structures, also corresponding initialization procedures are automatically generated.

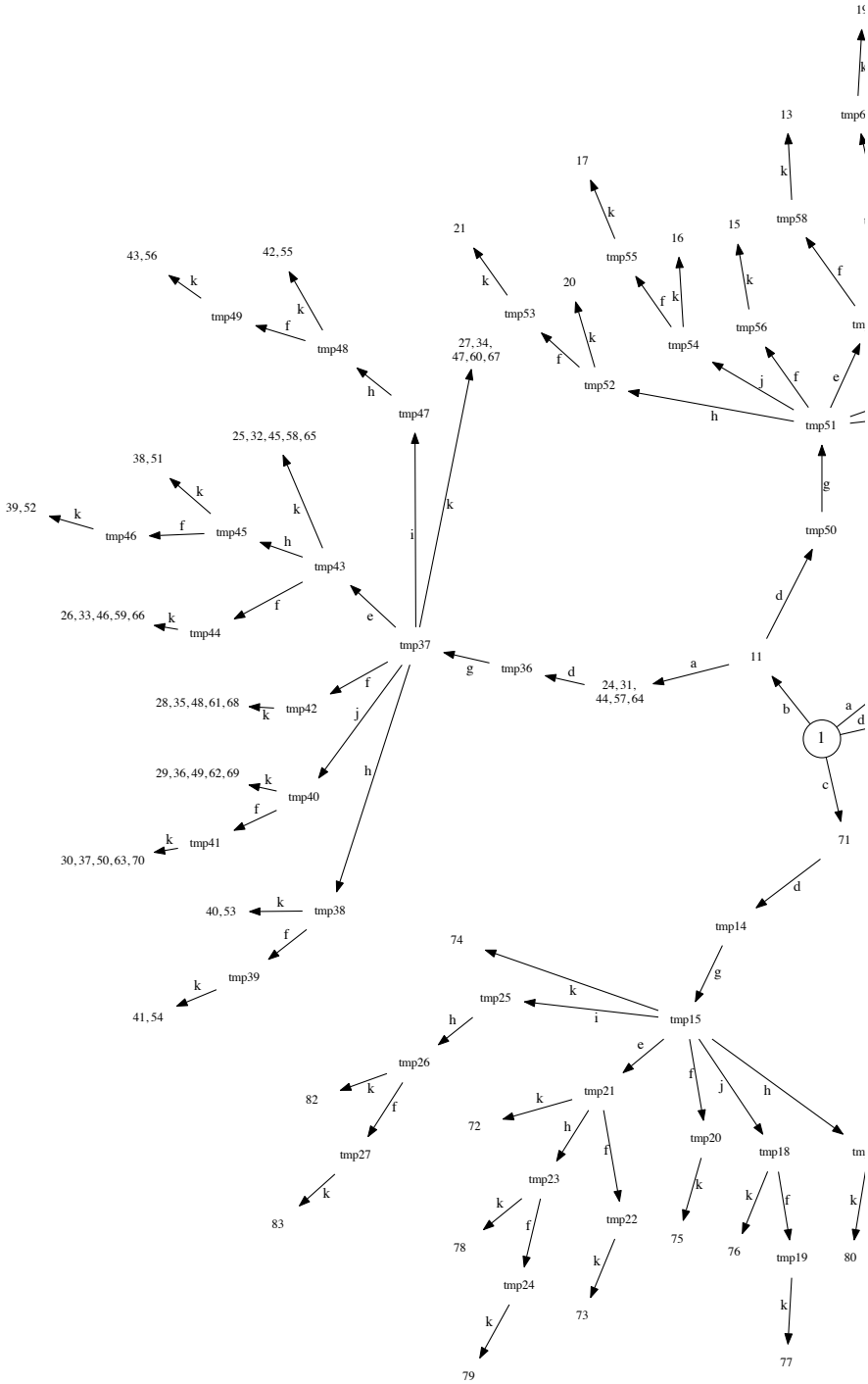
An initial characterization of the search space of principal forms can be obtained by looking at execution times[3]. For a given matrix, we can measure the execution time for different variants and different values of k. The results of these experiments are visualized in order of ascending execution time. The visualization is limited to display the 200 best-performing experiments. Two of such plots are shown in Figure 10.2. The dark gray star denotes the fastest experiment and the light gray circle the 200th instance in the order. The ordered sequence of the experiments is shown by the arrows, which fade out with increasing execution time.

In our discussion of these two plots, we use the fact that variants from the same subtrees in Figure 10.1 have numbers that are near each other. So, in the plot a longer arrow is a jump to another subtree, generated from another orthogonalization, etc. Both plots show a very different structure. We do notice, however, that the fastest variant is in both cases from the same subtree, with the dark gray star being located around 115. Subsequent fastest variants are different for both matrices, as the arrows progress in a vastly different manner. Another artifact that becomes clear is that for the *Erdos971* matrix, variants from more different subtrees are within the 200 best results. For example, the variants in the region $[0, 10]$ are frequently hit for the *Erdos971* matrix, but less so for the *OPF_10000* matrix.

Further plots can be created for other matrices, sparse algebra kernels and architectures. However, the two plots that are shown already present a clear difference in best performing variants for two matrices. One of the main questions is whether a ranking of the variants on execution time for one configuration of matrix, kernel and parameters can be used to predict the ranking for another matrix, kernel and parameter configuration. In other words, whether two such rankings

[2] In this chapter, we did not consider loop blocking. If loop blocking would have been taken into account, a multitude of different combinations of these data structures would have been generated.

[3] The execution times were measured on an Intel Xeon 5150 CPU at 2.66 Ghz, with 16GB RAM, running Ubuntu Linux 10.04.4. To remove fluctuation from the results, the computation performed by each variant is repeated 10 times. The compiler that was used is gcc 4.4. Several matrices have been used for the experiments and these were obtained from the University of Florida Matrix Collection [28].


1
a
b
c
d
11
24, 31, 44, 57, 64
71
d
tmp14
g
tmp15
74
k
i
tmp25
h
tmp26
82
k
f
tmp27
k
83
e
tmp21
72
k
h
f
tmp23
k
f
78
tmp24
k
79
tmp22
k
73
f
tmp20
k
75
j
h
tmp18
k
f
76
tmp19
k
77
k
80
tm
13
tmp6
k
tmp58
17
k
tmp55
16
f
k
tmp54
15
k
tmp56
f
tm
21
k
tmp53
20
f
k
tmp52
27, 34, 47, 60, 67
j
f
e
h
tmp51
fg
tmp50
d
43, 56
k
tmp49
42, 55
k
f
tmp48
h
tmp47
i
k
25, 32, 45, 58, 65
38, 51
k
39, 52
k
tmp46
f
tmp45
k
h
tmp43
26, 33, 46, 59, 66
k
tmp44
f
e
tmp37
g
tmp36
d
a
f
28, 35, 48, 61, 68
k
tmp42
j
29, 36, 49, 62, 69
k
tmp40
f
k
tmp41
30, 37, 50, 63, 70
h
40, 53
k
tmp38
f
k
tmp39
41, 54

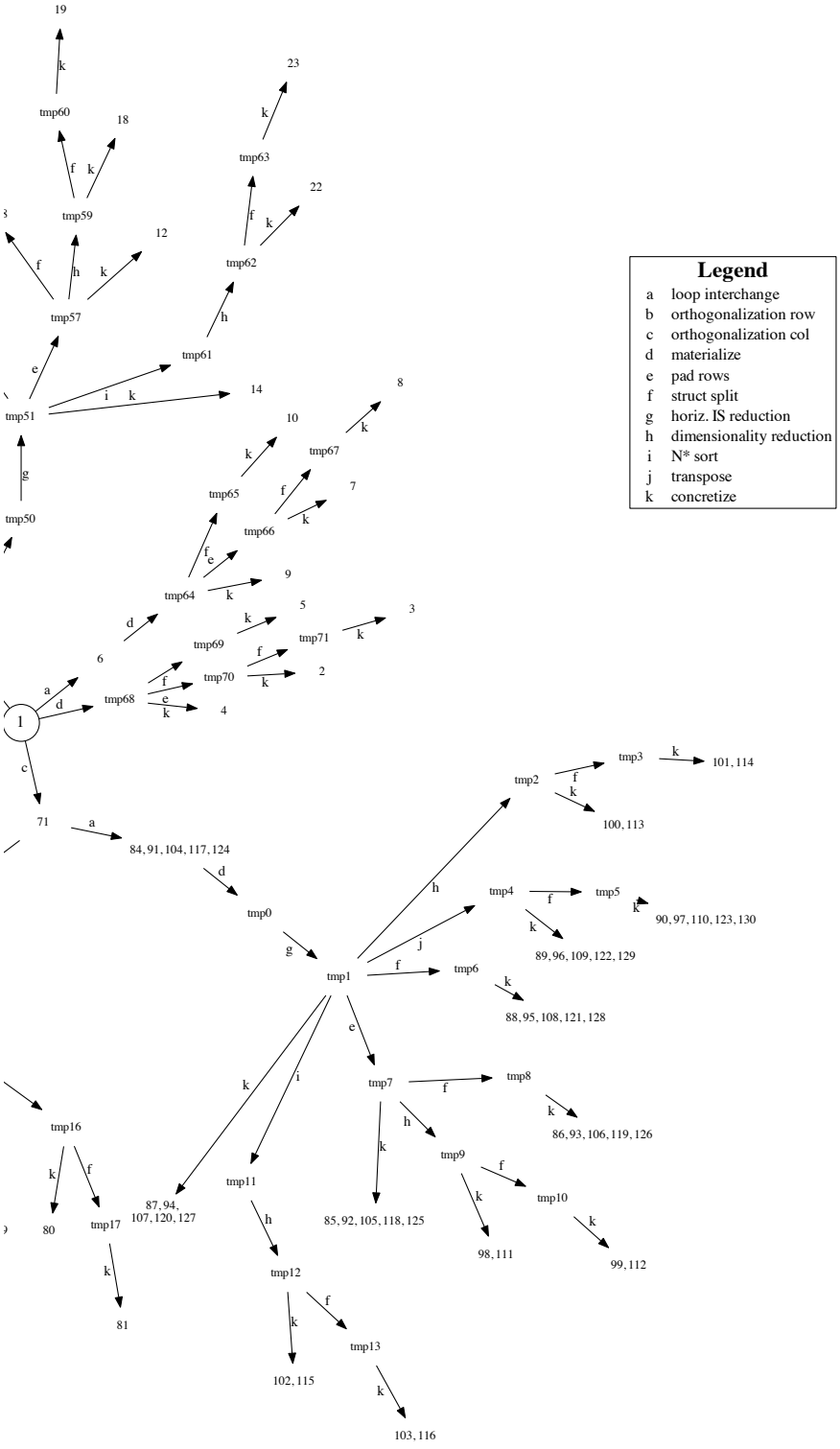


Figure 10.1: The full transformation tree of sparse matrix times k vector multiplication.

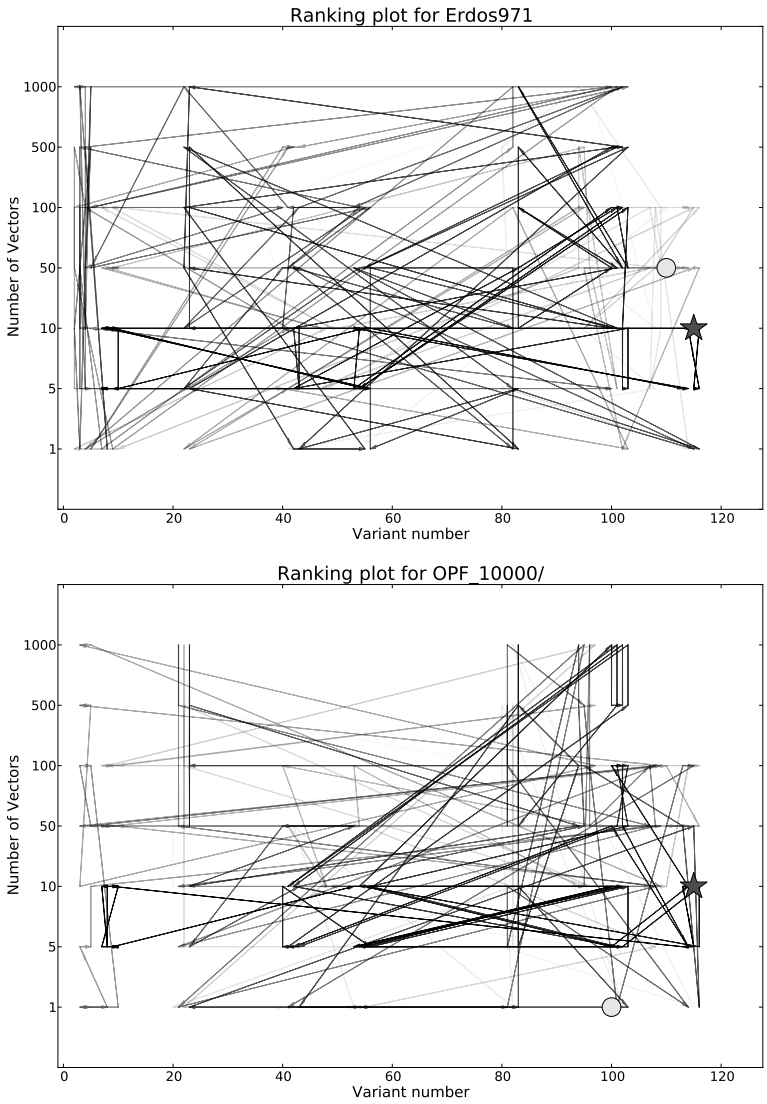


Figure 10.2: The plots show the 200 best-performing experiment instances, ordered in execution time from fastest to slowest. The star denotes the fastest variant, the circle the slowest.

bear any resemblance to one another. Instead of evaluating this graphically, we will quantify the differences in ranking in the next section, using the technique of Rank Correlations from the field of statistics.

10.4 Rank Correlations

As could be seen in the previous section, already the initial search space consisting of the principal forms is rather erratic. Although the plots give a good first impression of the transformation search space, these are not sufficient to come to a satisfactory quantification of the irregularity. In this section, we will use the Rank Correlation Coefficient as a measure for the relationship between two rankings.

Let a ranking be a permutation of a subset of n natural numbers: $\{i \mid i \in \mathbb{N} \wedge i \in [1, n]\}$. Given two rankings with the same n, a Rank Correlation Coefficient indicates the similarity of these two rankings. The coefficient takes on a value in the interval $[-1, -1]$, where 1 indicates the two rankings are equal and -1 indicates the rankings are each others reverse. Different methods to compute a Rank Correlation Coefficient exist. In this chapter, we make use of Kendall's τ [49], which is easy to compute. Computation of the coefficient is carried out by scoring the ranking. The score is computed by determining whether each pair of numbers in the ranking sequence is in the same order compared to the other ranking. For every in-order pair 1 is added to the score, for every out-of-order pair 1 is deducted from the score. The final score is divided by the maximum possible score (all pairs are in-order) to obtain the rank correlation coefficient.

The goal is to quantify the relationship of the ranking of variants for two experiment instances, for which the matrix, k and architecture are specified. The quantification is performed for the n best performing variants. To be able to compute the Rank Correlation Coefficient, a numerical ranking in $[1, n]$ is necessary for every variant in both experiments, so for the union of the variants found in both experiment instances. However, it can be the case that a variant from the n best performing variants for matrix A is not part of the best performing variants for matrix B. For such variants, a ranking is not specified for matrix B. This problem is resolved by assigning such variants a ranking of $n + 1$. In summary, best performing variants for a matrix A, which are not contained in the intersection of best performing variants of matrix A and B, are assigned a ranking of $n + 1$ for matrix B.

In Figure 10.3 the correlations of the rankings of the 10 best variants of experiment instances of the same matrix, but different k, are quantified. The line $y = x$ indicates the symmetric axis in this picture. From this figure it is observed that the correlation coefficient ranges from -0.4 to 0.5. That is, the majority of points are closer to 0, indicating a very weak correspondence between the rankings.

Another perspective is shown in Figure 10.4, where n best variants is varied and the rank correlation is shown between a number of vectors on the y axis and 50 vectors at the top, 10 vectors at the bottom. The plot quantifies the relationship between a ranking for 50 (or 10) vector multiplications and y vector multiplications. Note that the results for 50 versus 50 (and 10 vs. 10) have been set to 0 for clarity. For the *OPF_10000*, a clear relationship is seen between the ranking for 50

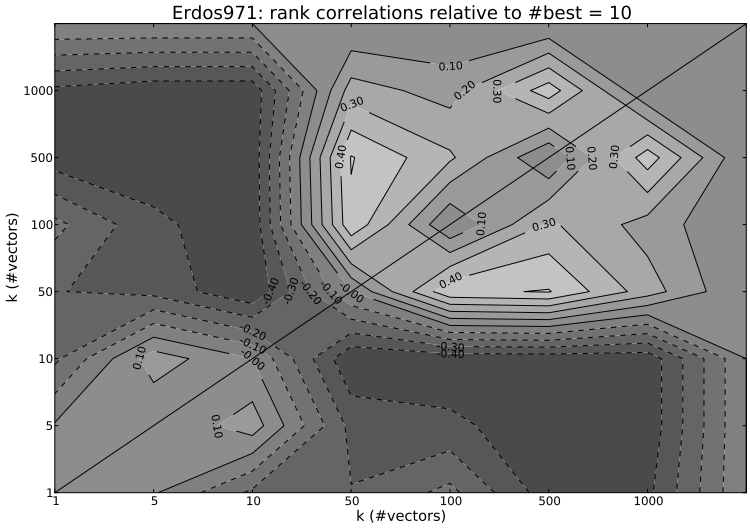


Figure 10.3: Quantification of the ranking correlation between a number of vectors x and y multiplied with the *Erdos971* matrix, considering the 10 best variants. Note that the values for $x = y$ are set to 0 and the symmetry of the figure is highlighted by the line $y = x$.

vectors and 100 vectors. However, for other values of k this relationship is weak. In the *Erdos971* plot, there is a strong correlation with $k = 5$ if the number of best variants considered is increased. A similar, but not as strong, trend is found for $k = 50$. Interesting is the clear reverse correlation area in the top-left corner of the plot. The 10 best variants at $k = 10$ are not included in the best variants for $k = 500$.

The quantification presented in this section shows that the transformation search space is indeed erratic. The lack of relationship between the rankings of different instances of a matrix, and a number k makes it very hard, if not impossible, to predict which variant would be optimal for a given instance. As a consequence, optimization must be carried out by performing an exhaustive search through this transformation search space.

Note that the transformation search space that has been described up till now *does not* contain any parametric compiler optimizations such as loop blocking and loop unrolling. Rather, the transformations that have been described affect the order in which computation is performed and the manner in which the input data is organized. In fact, application of parametric compiler optimizations is to be done next to the transformations that have been described so far. The exploitation of this parametric optimization search space is the second stage in our approach. We will show in Section 10.6 that by considering the transformation search space combined with a parametrized search space just containing Loop Unrolling, we are able to generate codes that achieve better results than existing library implementations of sparse algebra routines.

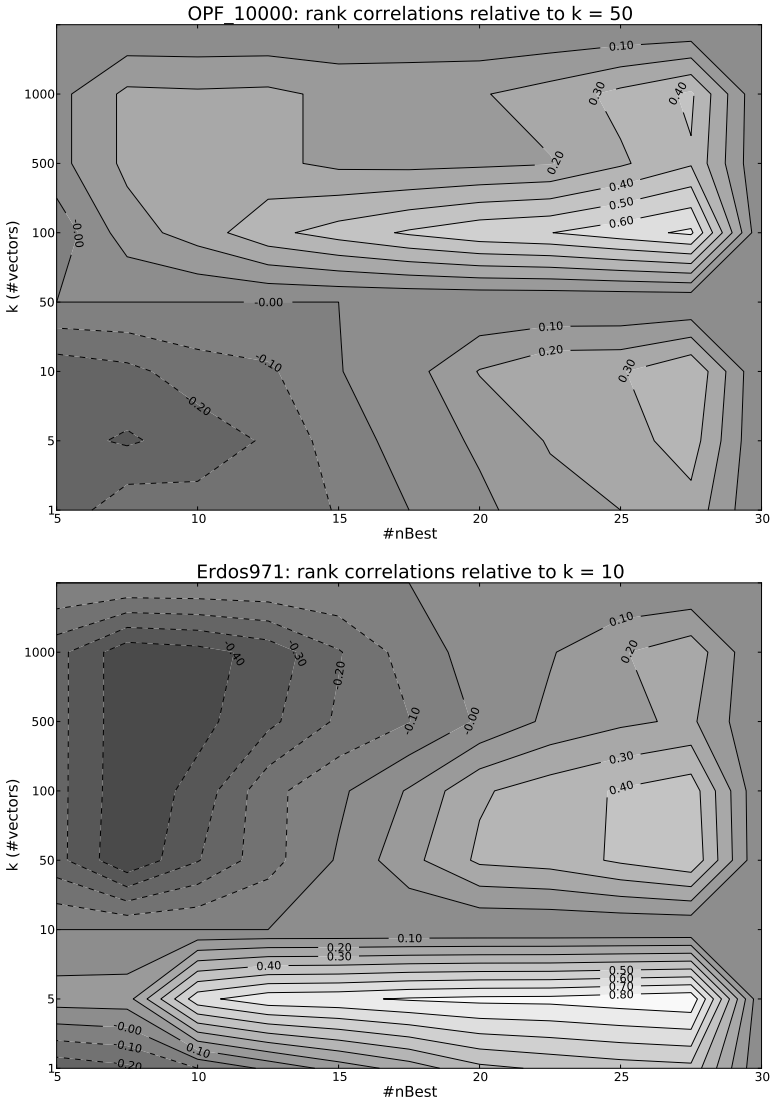


Figure 10.4: Quantification of the ranking correlation between a base number of vectors (50 at the top, 10 at the bottom) with increasing number of best variants being considered. Note that the values for y equal to base are set to 0 for clarity.

10.5 Irregularity Of Other Kernels

In the preceding sections, we have focused on the sparse matrix times k vectors kernel. We have also conducted experiments with two other sparse matrix kernels: sparse matrix matrix multiplication and lower triangular solve for unit matrices. The routines for sparse matrix times k vector multiplication and sparse matrix matrix multiplication have been instantiated with a similar transformation process of the code and data storage format. Because of this, the transformation search space is similar to Figure 10.1. However, for the triangular solve kernel, the transformation space is decreased from 130 to 76 variants, as can be seen in the transformation tree depicted in Figure 10.5.

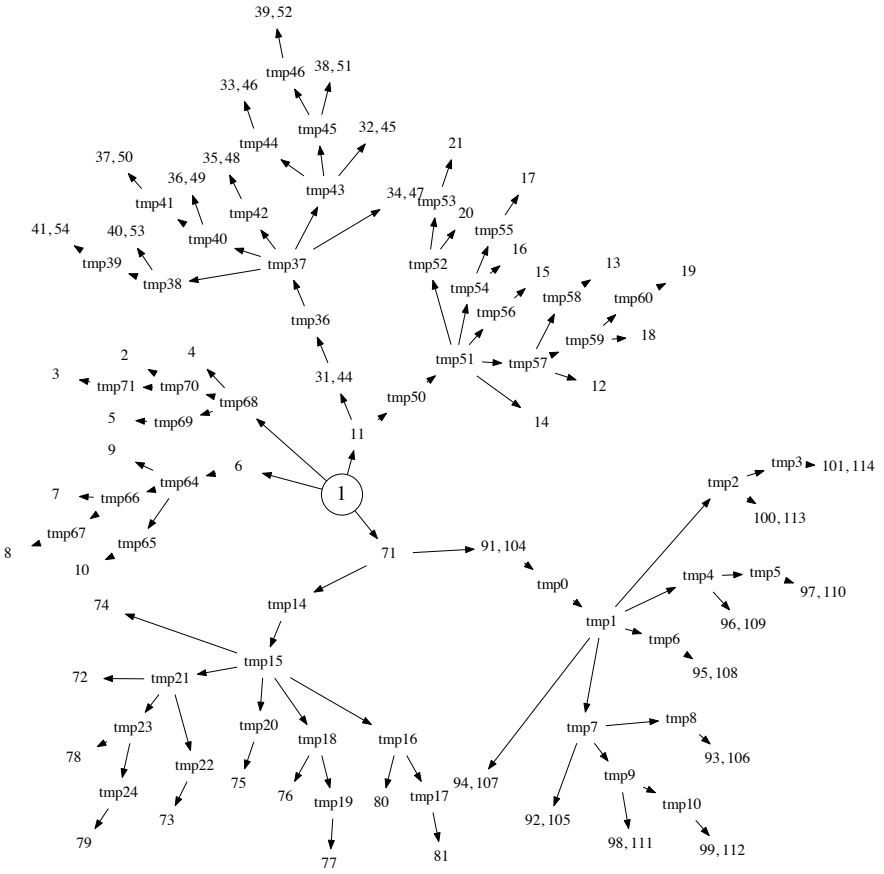


Figure 10.5: The full transformation tree of triangular solve.

Also in the case of sparse matrix matrix multiplication and triangular solve, the ordering of best-performing variants is highly irregular, as can be seen in Figure 10.6. These plots can be compared with the plot at the top of Figure 10.2,

which concerns the same matrix. This comparison shows that for the same matrix benchmarked on the same architecture, different kernels show vastly different best-performing instantiations.

10.6 Comparison To Existing Sparse Computation Libraries

In this section, a comparison is presented of the performance of sparse matrix codes generated with our framework to existing sparse algebra libraries. A detailed comparison is given for the sparse matrix times vector multiplication kernel with $k = 1$, because this routine is present in every sparse algebra library. The codes that have been generated by our framework follow from an exhaustive search through the described transformation search space, combined with a search through different possible Loop Unroll parameters.

The comparison comprises the following sparse algebra libraries: Blaze 1.2 [43], with the matrix stored in both row-major and column-major order; MTL4 [40], with the matrix stored in both row-major and column-major order; SPARSE1.3 [59]; SparseLib++ 1.7 [32], with the matrix stored in coordinate storage format, compressed row storage format and compressed column storage format.

The experiments have been performed on two architectures. The first architecture has already been introduced in this chapter and consists of an Intel Xeon 5150 CPU at 2.66 Ghz, with 16GB RAM, running Ubuntu Linux 10.04.4. The second architecture is a machine that consists of an Intel Xeon E5-2650 CPU at 2.00 GHz, with 64 GB RAM, running CentOS 5.0. The compiler used in both cases is gcc 4.4. These architectures will be referred to as the Xeon 5150 and Xeon E5 architectures respectively. Twenty matrices have been surveyed, taken from the University of Florida Matrix Collection [28]. To remove fluctuation from the results, the computation performed by each variant or library is repeated 10 times.

The results for the Xeon 5150 architecture are shown in Figure 10.7. In this figure, the result of the smaller matrices and the results of SPARSE1.3 have been omitted for the sake of legibility. The matrices along the x-axis have been ordered in increasing execution time for the code generated with our framework. The execution time is arranged along the y-axis. From the figure can be observed that although the row-major order variant of the different libraries is competitive in performance, the code generated using our approach is always faster.

Figure 10.8 shows the speedup in execution time of the code generated by our approach over the best-performing implementation from the existing sparse algebra libraries. For the majority of instances, our approach realizes at least a factor 1.1 speedup in performance. In some cases, such as *G2_circuit* and *Raj1* for the Xeon 5150, and *or2010* for the Xeon E5, the code generated using our framework is significantly faster, achieving speedups up to a factor of 1.9.

10.6.1 Search Space Reduction for Loop Unroll Optimization

In this section, we presented the results attained by our approach that followed from an exhaustive search of the transformation search space and the parametrized

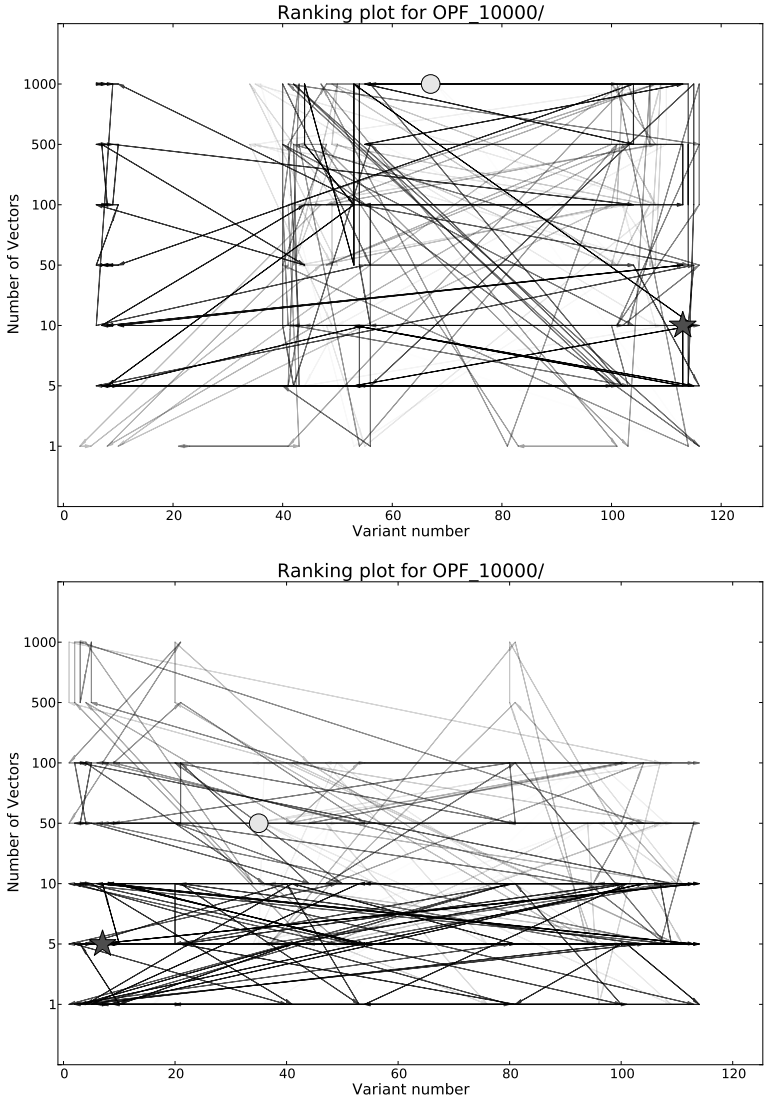


Figure 10.6: The plots show the 200 best-performing experiment instances for the sparse matrix matrix multiplication (above) and triangular solve (below) kernels.

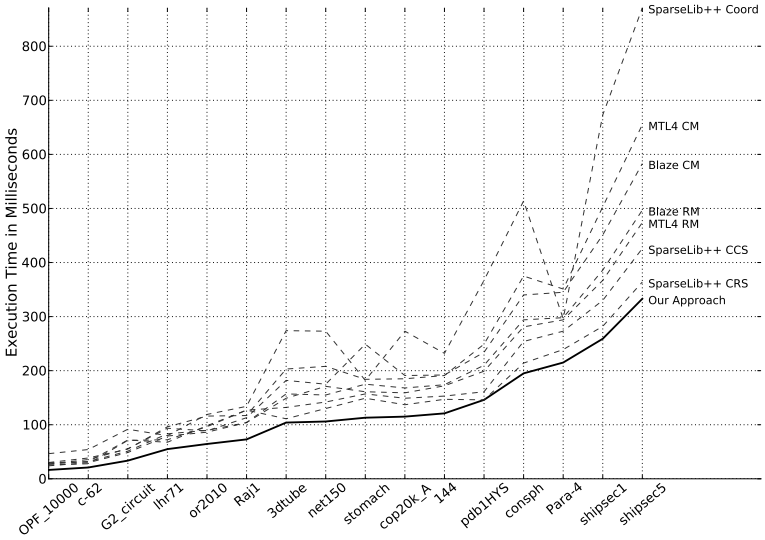


Figure 10.7: Comparison in execution time of automatically instantiated sparse matrix times vector computation and the different sparse algebra libraries on the Xeon 5150 architecture.

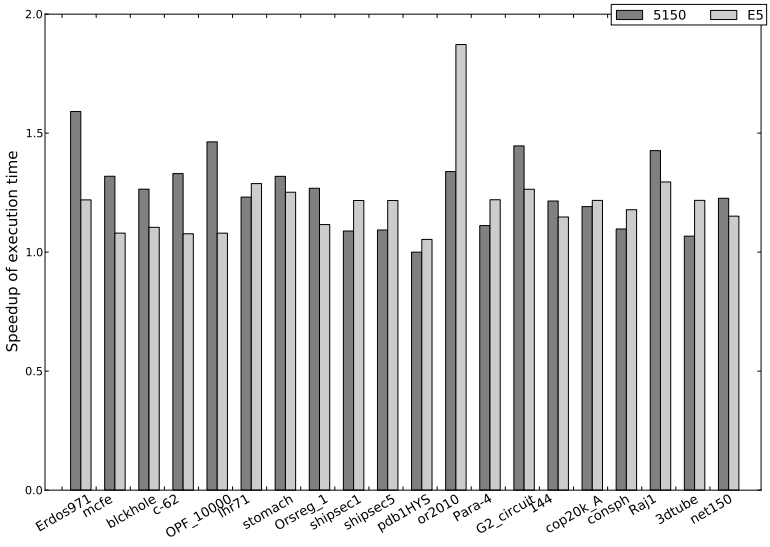


Figure 10.8: Speedup in execution time of the automatically instantiated sparse matrix times vector computation compared to the best-performing implementation from the existing sparse algebra libraries.

optimization search space. So far, only Loop Unrolling has been considered in the parametrized search. When other optimizations are included in the search, such as loop blocking, the size of the search space will grow substantially. Therefore, it is important to find methods to reduce the parametrized search space.

An often used technique for finding good parameter settings for parametric optimizations is Iterative Compilation [54, 34]. Note that Iterative Compilation is constrained to what we have described as the parametrized search space. In our case, techniques are needed for finding instances that are optimal with respect to the transformation search space and parametrized search space. For example, a variant that is not optimal within the transformation search space, may become the optimal variant after searching through the parametrized search space. The technique must be able to deal with these "cross-overs".

Figure 10.9 displays the relationship between the percentage of performance an instance is distanced from the optimum for an unroll level of 1 and the global optimum, considering all unroll levels. The execution time of the instances on the Xeon E5 architecture are shown. The lines represent different matrices. In the figure can be seen that variants at an unroll level of 1 must be searched up to about 28% distanced from the optimal performance found for an unroll level of 1, in order to find the global optimum. For the majority of matrices surveyed, however, the global optimum will be found by exploring the variants with a performance within 18% of the optimum for an unroll level of 1. This is a good indicator that it is not necessary to search the full parametrized search space to be able to eliminate the problem of cross-overs.

The extent of the parametrized search space that must be searched in order to find a global optimum is shown in Figure 10.10. Because the variants for an unroll level of 1 must always be searched, the extent of the search space that must be explored is at least 12.5%. For all matrices surveyed, it is only necessary to explore up to 19.0% of the parametrized search space in order to find a global optimum. This is a significant reduction compared to an exhaustive exploration of the search space, indicating the feasibility of pruning the parametrized search space in our approach.

10.6.2 Other Kernels

We also conducted the above described experiments on two other kernels: sparse matrix times matrix multiplication (with a 100-column dense matrix) and lower triangular solve with unit matrices. These experiments have been conducted on the Xeon 5150 architecture. The comparison with sparse matrix matrix multiplication from existing sparse algebra libraries has only been carried out with Blaze and MTL4, because SPARSE1.3 and SparseLib++ did not contain API for this computation. The *SPMM* column of Table 10.1, reports the speedups attained by the generated routine over the fastest implementation from an existing library. In all cases, a speedup of at least 1.17 is achieved, which is a decent improvement. For several matrices, the generated routine achieves an impressive speedup beyond a factor of 2 up to a speedup of a factor of 2.37.

An implementation of triangular solve is only found in the MTL4 and SparseLib++ libraries. The results are shown in the *TrSv* column of Table 10.1.

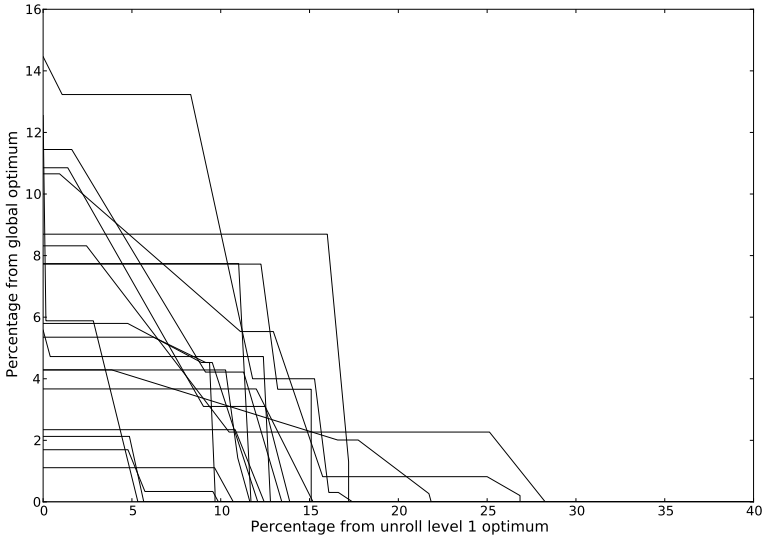


Figure 10.9: The relationship is shown between the percentage of performance an instance is distanced from the optimum for an unroll level of 1 and the global optimum, considering all unroll levels. The lines represent different matrices.

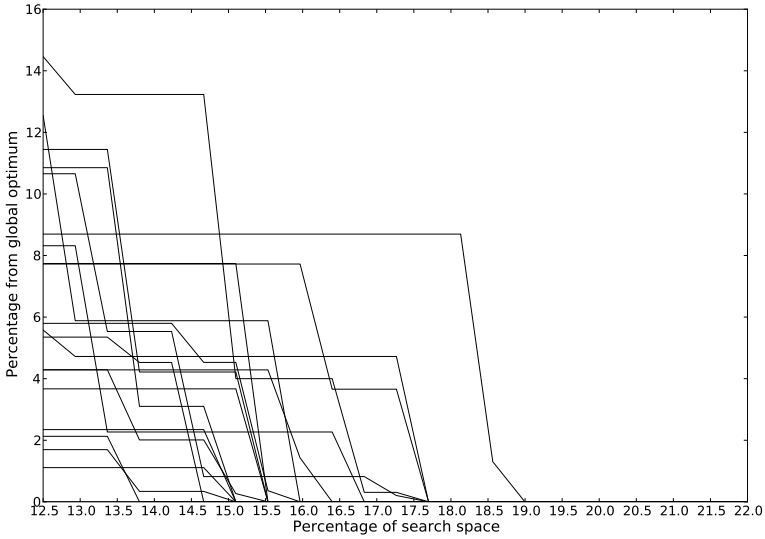


Figure 10.10: This figure illustrates the extent of the parametrized search space that must be searched to find a global optimum. The lines represent different matrices.

	SPMM	TrSv		SPMM	TrSv
Erdos971	2.28	2.11	pdb1HYS	1.36	1.04
mcfe	1.62	1.09	or2010	1.36	1.45
blckhole	2.31	1.01	para-4	1.42	0.98
c-62	1.44	1.12	G2_circuit	1.54	1.47
OPF_10000	1.59	1.17	144	1.17	1.01
lhr71	1.45	1.12	cop20k_A	1.42	1.09
stomach	1.59	1.18	consph	1.32	0.97
Orsreg_1	2.37	0.98	Raj1	1.46	1.35
shipsec1	1.50	0.99	3dtube	1.49	1.04
shipsec5	1.48	1.03	net150	1.35	1.05

Table 10.1: Speedup in execution time of the sparse matrix matrix multiplication and triangular solve kernels generated using our approach compared to the best-performing implementation from the existing sparse algebra libraries on the Xeon 5150 architecture.

In the majority of cases, a speedup is realized by the automatically generated routines, as big as a factor of 1.59 to 2.21 for a number of cases. For four cases however, a slight decrease in performance is seen. The decrease is so slight that the performance of the automatically generated routine is on par with the fastest library routine found.

In the previous subsection it was shown that it is only necessary to search up to 19% of the parametrized search space to find a global optimum. Similar results are found for the two kernels discussed in this subsection: up to 17% of the search space needs to be searched for sparse matrix matrix multiplication and up to 16% for triangular solve. So, also for other kernels it is feasible to significantly prune the search space.

10.7 Conclusions

In this chapter, we have described our approach for the automatic instantiation of efficient sparse algebra routines. This instantiation is combined with the reassembly of the original sparse matrix data structure into a form that is better aligned with the computation performed by the instantiated code. The transformations that lead to these different instantiations set up a large transformation search space. We have shown this search space to be very erratic and as a consequence no manageable subset of best-performing implementations can be chosen to be collected in a library. Rather, optimization must be carried out by performing an exhaustive search through this transformation search space.

We have shown that in addition to this transformation search space, parametric optimizations, such as loop unrolling and loop blocking, must be considered. These parametric optimizations set up a parametrized search space, further expanding the size of the search space as a whole. The size of the resulting search space is two orders of magnitude larger than search spaces derived in Iterative Compilation. Evidence has been presented that the search through the parametrized search space can be pruned to 20%, while a global optimum is still found.

Finally, we have shown that when this search space is properly exploited this results in variants of the computation that are faster than the implementations found in existing sparse algebra libraries, or at least equivalent in performance. Based on this, we argue that in order to get the maximum performance out of an architecture one cannot count on existing libraries, but an exhaustive search through a series of automatically instantiated routines must be performed.

CHAPTER 11

Handling Data Dependencies In The Forelem Framework

11.1 Introduction

Irregular computations are characterized by non-strided, unpredictable memory access. This defeats hardware features such as caching based on spatial locality and memory pre-fetching. Commonly, these irregular accesses are caused by the use of pointer-linked data structures or, even worse, a data structure that is unsuitable for the computation that is carried out. Static compiler analysis breaks down on pointer-linked data structures, as the order in which pointer-linked entities are accessed cannot be determined at the time of compilation. As the compiler cannot analyze the access pattern, it is refrained from applying effective techniques to optimize memory access or to parallelize the program. Because compilers can in most cases not exert extensive control on how data is stored, barely any opportunity is left to improve on this situation.

To overcome these problems, a different programming and optimization paradigm is needed. In this paradigm, irregular computations should be expressed in a different manner, breaking down the used data structures and capturing the essence of the computation. This will expose more information to the compiler about the order in which data is visited and dependencies between data that were encapsulated in data structures. Furthermore, this paradigm must enable the compiler to restructure the storage of data, next to restructuring the computation. As a result, the number of opportunities to effectively optimize and parallelize the computation will be greatly increased.

In this chapter, the *forelem* framework is extended so that data dependencies can be handled properly allowing irregular computations to be fully optimized and parallelized. Irregular computations can be naturally described within the *forelem* framework as can be seen in the previous chapters. Any data structure that is used by the computation is reduced to tuples. Dependencies that are encapsulated in loop nests are made explicit as dependencies between tuples. Com-

putations to be carried out are expressed as the iteration of a (sub)set of tuples and an operation based on these tuples. The major benefit of expressing computation on the elementary tuple level, rather than on the (complicated) data structure level, is that any obstructions introduced by the use of complicated data structures or unnatural encoding of dependencies are eliminated. As a consequence, the compiler is provided with more opportunities to automatically optimize the code. Furthermore, the compiler can explicitly control the way data is stored by modifying the structure and organization of the tuples. This leads to modifications to the data structure(s) used by the computation and, hence, the actual data structure(s) are constructed during the code generation phase.

Because dependency information is made explicit as dependencies between tuples in the tuple pool on which is operated, it is trivial to deduce which operations on tuples can be executed at the same time. This makes the proposed extension to the *forelem* framework especially suited for the automatic parallelization of codes. Using this extension, the central action for parallelization is to map a given specification of the computation onto an execution model. Two execution models will be described: affine embedding and static scheduling. The *forelem* framework that is extended is equipped with transformations that can be applied before and after the mapping onto a particular execution model, or execution scheduling. These transformations can be applied in many different ways, giving rise to a large optimization space. Effective application of these transformations leads to the generation of codes that are competitive with hand-optimized codes.

To take advantage of this paradigm, it is not necessary to rewrite existing codes to codes that operate on tuples. Rather, current implementations of irregular computations in for instance the C programming language can be automatically mapped into the tuple-based programming model. Consequently, sophisticated parallel codes can be generated from a starting point provided in the C programming language. We will demonstrate that from an ordinary triangular solve code written in C, parallelized implementations can be automatically produced that up till now could only be derived by hand. The performance of these automatically generated implementations is comparable to that of hand-optimized implementations.

This chapter is organized as follows: Section 11.2 introduces the programming model in which generic computations can be expressed in terms of tuples. Section 11.3 discusses how tuple-based expressions of computations are mapped onto an execution model. In Section 11.4 transformations are described that can be applied before and after execution scheduling. Section 11.5 discusses the applicability, versatility, university and transferability of the proposed programming model. In Section 11.6 a case study is presented, in which through the use of the proposed extension of the *forelem* framework implementations of triangular solve are derived from a starting point written in the C programming language. Section 11.7 presents our conclusions and plans for future work.

11.2 Handling Data Dependencies Between Tuples

In this section, the expression of dependencies between tuples is discussed. A method is introduced to explicate the dependencies between tuples in the tuple iteration structure, allowing for irregular applications to be naturally expressed in terms of loops processing tuples. First, we will give a small review on how tuples are handled by the *forelem* framework.

11.2.1 Iteration of Tuples

Data to be processed is specified as (multi)sets of tuples. The computation is expressed in terms of loops processing the tuples. Different transformations are implemented in the framework, ranging from standard compiler optimizations, such as Loop Interchange [4], Loop Fusion [52], Scalar Expansion and Def-Use analysis [2, 50], to transformations that address the order in which tuples are executed and stored.

The principal syntactic construct in the *forelem* framework is the *forelem* loop. A *forelem* loop iterates (a subset of) a multiset of tuples and performs an operation on these tuples. As an example, consider a multiset `T` containing tuples with fields `field1` and `field2`: `(field1, field2)`. Then, the following loop sums the values of `field1` of tuples of which `field2` equals the value `9`:

```
sum = 0;
forelem (i; i ∈ pT.field2[9])
  sum += T[i].field1;
```

Iteration of the *forelem* loop is controlled with the "index set" `pT.field2[9]`, which in this case contains all subscripts into `T` for tuples of which `field2` equals `9`. The index set specifies which tuples will be visited, but does not specify the order in which these tuples are visited, which is undefined.

Naturally, *forelem* loops can also be nested. The value of a tuple in a tuple pool can be used to access tuples in another tuple pool, say `S`:

```
forelem (i; i ∈ pT.field2[9])
  forelem (j; j ∈ pS.field1[T[i].field1])
    sum += S[j].field2;
```

The index set conditions are designed such that they can be rewritten to a conditional clause of *if* statements. This property is used to rewrite loop nests into a form with the conditions tested in the innermost loop, enabling a variety of loop transformations to be performed. When this is done for the above loop nest, the result is:

```
forelem (i; i ∈ pT)
  forelem (j; j ∈ pS) {
    if (T[i].field2 == 9 &&
        S[j].field1 == T[i].field1)
      sum += S[j].field2;
  }
```

This loop nest will produce equivalent results, since the statement in the inner loop is executed for the same set of tuples from T and S. By moving the conditions to the innermost loop, it has been made possible to apply the loop interchange transformation, after which the conditions can be moved back from the inner loop to the corresponding index sets.

11.2.2 The Ready Clause

The most important property of the *forelem* loop construct is that through the use of index sets the tuples that should be visited are specified, but not in which order. Iteration of the selected tuples may happen in any order. In other words, *forelem* loops are inherently parallel.

For irregular codes, this is a problematic property, as statement instances in an irregular code commonly have a dependency on another statement instance to be executed first. In the case of linked list traversals, a certain element must be visited before it is known what the next, or previous, item to visit is. Matrix computation codes, such as triangular solvers, need to ensure writes to rows $k \in [0, i)$ are completed before column i can be processed.

To accommodate the specification of such dependencies, we propose to extend the *forelem* framework, or in particular the index set capabilities, with a *ready* clause. The *ready* clause is an expression that for a given tuple `t` in tuple pool `T` specifies which tuples `r` in tuple pool `T` must have been visited. Using this method, dependencies can be set up between tuples in a tuple pool. As will be discussed in Section 11.5.1, a bijection can be set up between the iteration space of an original loop and tuples, which enables the *ready* clause to express dependencies between statement instances as well.

The *ready* clause naturally extends the index sets that are used to control iteration in *forelem* loops. An example of the syntax for this clause is:

```
forelem (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
  SEQ;
```

where SEQ denotes a sequence of statements and $\nabla(\mathtt{T[r]}) = \mathtt{T[q]}$ is the *ready* expression. For a more formal treatment of the *ready* expression, see Section 11.3. The index set `pT` will contain subscripts `q` into tuple pool `T`, for which all tuples `T[r]` in `T` that meet the specified *ready* condition have been processed. Additionally, the subscript `q` may not have been processed already. As a definition, this evaluation takes place before the *forelem* loop is entered. As a consequence, no further subscripts will be added to the index set while the loop is in progress, including new tuples that have become ready after any modifications to `T` that may have occurred in the body of the *forelem* loop.

An example of a *ready* expression is `T[r].z == T[q].y` (see Section 11.3 how this can be expressed formally), which specifies that in order to be able to process `T[q]`, all tuples `T[r]` must have been visited that have a `z` field equal to the `y` field of `T[q]`. In general, in *ready* clause expressions, the tuples addressed by `r` and the iterator variable of the loop in which the clause is embedded (in the case of the example `q`) are used as operands, and standard C operators such as ==, !=, ||, && and ! are used as operators.

Note that the specification of dependencies in this manner allows the compiler to find a suitable execution schedule for the computation at hand, without being bound to redundant constraints. This is contrary to existing approaches, where the dependencies are encoded in a particular nesting and ordering of loops. In that case, a compiler may modify the loop nesting and ordering, as long as any dependencies in the loop are not broken. So, the compiler has to deduce the actual dependencies from an encoding in the loop structure and may find redundant dependencies that are an artifact of encoding the actual dependency in this structure.

11.2.3 Tuple Marking

For a given tuple, the *ready* construct specifies a condition for tuples that must have been processed already. This implies that the possibility must exist to make a distinction between tuples that have been processed/visited and tuples that have not been processed/visited.

To be able to make this distinction, we introduce the possibility of "marking" tuples in a tuple pool. The following operations are defined:

1. `reset(tuple_pool)`. Resets all marks in the given tuple pool.
2. `mark(tuple)`. Mark the specified tuple.
3. `unmark(tuple)`. Unmark the specified tuple.
4. `marked?(tuple)`. Returns whether the specified tuple is marked.
5. `unmarked?(tuple_pool)`. Returns whether the tuple pool contains any unmarked tuples.

As we will see in the next subsection, with these operators it is possible to come to a formalization of how *forelem* loops with a *ready* clause are executed.

11.2.4 Specification of the ready clause

So far, a number of requirements for the execution of *forelem* loops with a *ready* clause have been put forward. An index set containing a *ready* clause is evaluated before the *forelem* loop is entered. Through this requirement, it is guaranteed that no new tuples can become ready during execution of the loop. A second requirement is that only subscripts into a tuple pool `T` are considered that have not yet been visited and are thus not marked.

Similar to the ability to move the conditions tested in index sets to the inner loop, it is possible to move the testing of the *ready* inside the loop. This must be done while taking the requirements for the execution of these loops into consideration. By making use of the tuple marking operations, these requirements can be met and the result is a formal specification of how *forelem* loops with a *ready* clause are executed:

```
visited = ∅;
forelem (q; q ∈ pT)
  if (!marked?(T[q]) && ready(r)[∇(T[r]) = T[q]]) {
```

```
    SEQ;
    visited = visited ∪ q;
  }
for (q'; q' ∈ visited)
  mark(T[q']);
```

The first loop is a regular *forelem* loop which visits all subscripts of pT once and tests whether these are marked and are ready for execution. This is equivalent to a loop which computes which subscripts are to be visited before execution of the actual loop. The actual execution of the statements SEQ has been merged into this loop. Note that the requirement to disallow new tuples to become ready during execution of the loop, resulting in these being added to the index set, is enforced by updating the marks for tuples visited in the loop after this loop has completed execution.

Note that this is only a formal, algebraic, specification of how *forelem* loops with a *ready* clause should be executed. This specification is necessary to be able to define transformations on these loops. In practice, loops are not executed in this manner, rather the code is transformed to a suitable execution model as will be discussed in Section 11.3.

11.2.5 Ensuring All Tuples Are Processed

Commonly, when a *ready* condition is present not all tuples of a tuple pool T are to be processed in a single execution of the loop. Rather, a execution of the loop will ready subsequent tuples, that are to be visited in subsequent executions of the same *forelem* loop. So, to process all tuples in a tuple pool T (provided a ready condition is specified such that execution of all tuples can indeed be reached), a *while* loop is to be added. The tuple marking as described above ensures that tuples are only visited once, even when the *forelem* loop is executed multiple times:

```
reset(T);
while (unmarked?(T))
  forelem (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
    SEQ;
```

As a shorthand for this pattern, a *whilelem* notation is introduced:

```
whilelem (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
  SEQ;
```

so, when iteration of the index set is completed, the index set is re-instantiated as long as T contains unmarked nodes.

In fact, the *whilelem* construct can be viewed as an synchronized execution of all the tuples, where first all tuples are to be processed which can at first be processed, then all tuples will be processed which were enabled by the previous tuples, etc. One could also imagine a more dynamic implementation of *whilelem* which allows the execution of a tuple at any stage. More formally, let us call this *whilever* (whatever tuple can be processed is processed), and

```
whilever (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
  SEQ;
```

implies that at each iteration `mark[T[q]]` will be executed as well as a "new" evaluation of the *ready* clause. In this case, the number of possible execution orders is greatly enhanced. However, at the basis of the *forelem* loop concept is the assumption that the index set `pT` can be precomputed, thereby still allowing the index set to be precomputed whilst still allowing a random order of the tuples to be executed. Therefore, this *whilever* construct is not further elaborated on in this chapter. Also, note that whenever the number of tuples ready at each "stage" equals one, then *whilever* is equal to *whilelem*.

11.3 Execution Models

The *forelem* loop with *ready* clause provides an algebraic means to specify the computations that must be carried out on the tuples as well as the dependencies between these tuples. To come to an executable code, this abstract specification must be mapped onto an execution model. In this section, this mapping process is described together with two execution models: affine embedding and static scheduling.

Consider tuples `(field1, field2, ..., fieldn)` with `field1` $\in$ `F1`, `field2` $\in$ `F2`, ..., `fieldn` $\in$ `Fn`. The full tuple space `TS` that is set up by these tuples is defined as `F1`$\times$`F2`$\times ... \times$`Fn`. Let a tuple pool `T` be a subset of `TS`. Now, consider the following *forelem* loop:

```
forelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
  SEQ;
```

The expression in the *ready* clause gives rise to projections Δ_{in} and ∇_{out} that are defined as follows:

$$\nabla : \texttt{F1} \times \texttt{F2} \times ... \times \texttt{Fn} \rightarrow \texttt{F1}' \times \texttt{F2}' \times ... \times \texttt{Fm}$$

with $m \leq n$. The *ready* expression in the loop is then replaced as follows:

```
forelem (q; q ∈ pT.ready(r)[∇out(T[r]) = Δin(T[q])])
  SEQ;
```

in other words, all tuples `T[r]` that project onto the same set of tuples as Δ_{in}(`T[q]`) must have been visited before `T[q]` can be visited. In case Δ_{in} is invertible, then one obtains $\Delta_{in}^{-1}(\nabla_{out}(\texttt{T[r]})) = \texttt{T[q]}$ or in short $\nabla(\texttt{T[r]}) = \texttt{T[q]}$. This *forelem* loop is still an algebraic specification of the computation to be performed and can be executed through dynamic execution. In dynamic execution, the index set with *ready* clause is evaluated dynamically at runtime. In the two subsequent subsections, two execution models are now defined that reformat the *forelem* loop to a loop without *ready* clause. This reformatted loop can be subjected to further code transformations and is used as a starting point to generate efficient executable code. We refer to this process as "execution scheduling".

11.3.1 Affine Embedding

Whenever tuples contain integer fields, these tuples can be iterated by enumerating all possible values of these integer fields in all possible combinations. In affine embedding, an enumeration order of these integer values is determined that satisfies any *ready* condition, or tuple dependency, that is defined.

Given tuples `(x, y, z)` in a tuple pool `T`, where `y` and `z` are integer fields. The notation `T.y` denotes the set of all values `t.y` of all tuples `t` in `T`. So, `T.y` and `T.z` contain all values that occur for these fields in all tuples in the tuple pool. Suppose these fields have values within the interval $[0, N)$, then these fields have integer ranges `T.y` $\subset I_y = [0, N)$ and `T.z` $\subset I_z = [0, N)$.

These integer ranges must be transformed such that the tuples are visited in an order that does not violate the tuple dependencies. To accomplish this, a unimodular matrix U (see for instance [13, 99] for a treatment of the use of unimodular matrices for performing loop transformations) is defined such that:

$$\forall t, r \in T : \nabla_{out}(r) = \Delta_{in}(t) \Rightarrow U(r) <^l U(t)$$

where $<^l$ denotes lexicographical ordering. From U functions $f_y : I_y \rightarrow I_y$, $f_z : I_z \rightarrow I_z$ follow such that:

$$\forall t, r \in T : \nabla_{out}(r) = \Delta_{in}(t) \Rightarrow (f_y(r.y), f_z(r.z)) <^l (f_y(t.y), f_z(t.z))$$

Using these functions, an affine embedding can be written as follows:

```
for (i; i ∈ f_y(T.y))
  for (j; j ∈ f_z(T.z))
    SEQ;
```

As an example, consider a simple triangular solve loop:

```
whilelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
  B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

on tuples `(x, y, z)` containing values `(A[j][i], j, i)`, as will be described in Section 11.5.1. Consider

$$U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

which implies a loop interchange. Then it can be found that f_y and f_z are the identity function. To show that the *ready* condition is satisfied, consider tuples (y, z) (just the integer fields) at time t: (y, z, t). For all tuples holds that: (1) $y > z$, the tuple pool only contains the elements of the lower triangle; (2) for two tuples a, b, if $a.t < b.t$ then $a.y \leq b.y$, the y values denoting the row number `j` in `A[j][i]` are processed in order. The ready function is defined such that $\nabla_{out}((j, i)) = (j)$ and $\Delta_{in}((j, i)) = (i)$.

Now, for all tuples (k, m, t) holds that

$$\{(n, p, s) : s > t \wedge \nabla_{out}((n, p)) = \Delta_{in}((k, m))\}$$

is an empty set. Consider this is not the case, then $\exists(n,p,s) : s > t \wedge \nabla_{out}((n,p)) = \Delta_{in}((k,m))$. From the definitions of ∇_{out} and Δ_{in} we know that $n = m$. (1) gives $p \geq m$. From (2) we know that $n > p$. So, $n > p \geq m$. Contradiction.

As a result, we can write an affine embedding of the triangular solve loop as follows:

```
for (j; j ∈ f_y(T.y))
  for (i; i ∈ f_z(T.z))
    forelem (q; q ∈ pT.(y,z)[(j,i)])
      B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

Note that this is an intermediate representation of the loop nest. As will be further described in Section 11.6.1, many different variants of the triangular solve code can be generated from this loop nest.

11.3.2 Static Execution

Through a symbolic execution of a *forelem* loop with *ready* clause, it is possible to derive a static execution schedule in which the tuples can be visited without violating the *ready* clauses. This schedule can be stored and later be used to perform the actual computation, without disturbances caused by re-computation of the *ready* clause.

In this subsection, we will consider the following loop nest:

```
reset(T);
while (unmarked?(T))
  forelem (q; q ∈ pT.ready(r)[∇_out(T[r]) = Δ_in(T[q])])
    SEQ;
```

To be able to derive a static execution schedule from this loop, all values used in the ready clause may not be written to by `SEQ`. This enables a symbolic execution of the *forelem* loop. Alternatively, a copy of the tuple pool can be used. Any other field may be changed by the statements in `SEQ` without loss of generality.

The derivation is carried out by determining which subscripts `q` of the specified index set can be processed at the same time. In other words, the ready clause is satisfied for these tuples. Groups of tuples are formed that can be executed at the same time, which are called "levels". Tuples are tagged with the level they belong to, by adding a field name `level` to the tuple. This leads to the following loop, which tags all tuples with the correct level:

```
reset(T);
l = 0;
while (unmarked?(T)) {
  forelem (q; q ∈ pT.ready(r)[∇_out(T[r]) = Δ_in(T[q])])
    T[q].level = l;
  l = l + 1;
}
```

Note that the *forelem* loops within the *while* loop make use of the two important properties that have been defined in Section 11.2.2:

1. All visited tuples are marked after execution of the loop, such that no tuples that were newly made ready are visited.

2. The loop does not visit tuples that have been visited already.

After the tagging has been carried out, the loop performing the actual computation can be carried out as follows:

```
for (l' = 0; l' < l; l'++)
  forelem (q; q ∈ pT.level[l'])
    SEQ;
```

Due to the absence of a *ready* clause in this loop nest, no dynamic evaluation of the *ready* clause is necessary at run-time and the loop can be executed according to a static execution schedule.

11.4 Transformations For Ready Loops

Several transformations are defined in the *forelem* framework that target loop structure and arrangement of tuples. In this section, a number of transformations are described that are specific to *forelem* loops with a *ready* clause. These transformations can be divided into transformations that are applied before execution scheduling and transformations that are applied after execution scheduling.

11.4.1 Before Execution Scheduling

An index set with a *ready* condition contains subscripts to tuples in a tuple pool which have not yet been processed and are ready for execution. Due to the nature of the *forelem* loop, the tuples referenced by these subscripts may be executed in any order. In certain cases, it is useful to group sets of tuples with certain similar properties for execution before scheduling takes place. This influences the final execution schedule of this loop. In this section, the Projection transformation is proposed to accomplish this. By applying the Projection transformation on a given loop on different properties, a transformation space is set up yielding different variants of the same loop with different performance characteristics. The Projection transformation is defined as follows. Consider:

```
forelem (q; q ∈ pT.ready(r)[∇_out(T[r]) = Δ_in(T[q])])
  SEQ;
```

where in the *ready* expression the following operands are used: `T[q].field1`, `T[q].field2`, `...`, `T[q].fieldn`. For this particular loop, a grouped execution is defined as follows:

```
forelem (q; q ∈ pT.ready(r)[∇_out(T[r]) = Δ_in(T[q])])
  forelem (p; p ∈ pT.(field1, field2, ..., fieldn)
      [(T[q].field1, T[q].field2, ..., T[q].fieldn])
    SEQ;
```

To understand the validity of this transformation, it is important to note the following: if for a given tuple `t` in `T` all dependent tuples are ready based on `field1`, `field2`, ..., `fieldn`, then for all other `t'` in `T` with equal values for `field1`, `field2`, ..., `fieldn` the ready sets are satisfied. Therefore, it is valid to process all of these tuples in the inner loop. In this case, execution is grouped in groups of tuples having equal values for `field1`, `field2`, ..., `fieldn`.

Next to conditioning the values of `field1`, `field2`, ..., `fieldn` in the inner loop, conditions may be added to test on other properties of the tuples to define a further projection. Important is that the fields that are present in the *ready* expression may never be omitted from condition testing in the inner loop.

Because we have defined *forelem* loops with a *ready* condition to only mark tuples visited in the loop body as visited after the loop has finished execution, the transformation needs to make a further modification. To avoid processing tuples that are visited in the inner loop for a second time by the other loop, these tuples must be marked as visited immediately, or at least within the loop body of the outer loop.

With projection, tuples to be processed are grouped based on certain properties. This is similar to the Orthogonalization transformation [83] that is defined in the *forelem* framework. Orthogonalization imposes a specific iteration order on a loop, based on the values of one or more fields of the tuples. For example:

```
for (k; k ∈ N)
  forelem (q; q ∈ pT.y[k])
    SEQ;
```

executes *SEQ* for tuples with the same value for field `y`, with `k`'s value in increasing order. In fact, the tuples to be executed are grouped into groups of tuples with equal values for field `y`.

Although Orthogonalization seems similar to the Projection transformation that has just been proposed, there is a fundamental difference. In the orthogonalized loop, tuples are always ready for execution: there is no *ready* condition. To see why orthogonalization does usually not have the desired effect, consider the following:

```
for (k; k ∈ N)
  forelem (q; q ∈ pT.y[k].ready(r)[∇out(T[r]) = Δin(T[q])])
    SEQ;
```

Only tuples `t` ∈ `T: t.y = k` can be visited for which all dependent tuples are ready. This may not be the case for all tuples `t`. So, after a full iteration of the outer loop, not all tuples may have been processed. As a consequence, this loop nest must be surrounded by a *while* loop that repeats execution of this loop nest until all tuples have been visited. Naturally, as a result, the tuples are no longer processed in exact increasing order of `k`, defeating the goal of orthogonalization.

11.4.2 After Execution Scheduling

Loop nests that are the result of mapping a *forelem* loop with *ready* clause onto an execution model, can be subjected to further transformations. A number of

these transformations will be described for loop nests that are the result of static scheduling. An example is:

```
for (i = 0; i < L; i++)
  forelem (q; q ∈ pT.level[i])
    SEQ;
```

where L is the number of levels that have been identified during the computation of the schedule. Within this loop nest, execution of tuples contained in a certain level can again be grouped based on certain properties. To accomplish this, the Orthogonalization transformation can be used to group the tuples this example code on field y:

```
for (i = 0; i < L; i++)
  forall (j; j ∈ Nj)
    forelem (q; q ∈ pT.(y,level)[(j, i)])
      SEQ;
```

Since all tuples in a level can be processed in parallel, the loop with iteration j is made a *forall* loop to indicate all identified groups can be processed in parallel.

An important difference with the Projection transformation described in the previous section is that since Projection is performed before execution scheduling, the grouped tuples are placed in separate levels. When Orthogonalization is applied after scheduling, groups are identified within levels. Secondly, orthogonalization after scheduling is not bound to any restrictions on the conditions like is the case with Projection.

Another transformation that can be performed after scheduling is to ensure sets of tuples with a certain property are processed at the same time. For example, all tuples with equal values for the y field should be processed at the same time. However, due to the imposed *ready* clause these tuples may be spread among different levels. In order to still execute all tuples with equal y field at the same time, the highest level in which such a tuple is placed should be found. All tuples with equal y can be safely executed in that particular level.

This transformation can be implemented by analyzing and modifying the level tags on the tuples. A pseudocode to perform this modification is as follows:

```
forelem (j; j ∈ Nj) {
  max_level = 0;
  forelem (k; k ∈ pT.y[j])
    max_level = MAX(max_level, T[j].level)
  forelem (k; k ∈ pT.y[j])
    T[j].level = max_level;
}
```

11.5 Characteristics of the Extended Forelem Framework

The extended *forelem* framework that is proposed in this chapter has several compelling characteristics. These include *applicability*, as the techniques that have been

described can be applied to a wide variety of existing codes, *universality* as all data structures can be represented in terms of tuples, *transferability* allowing sparse codes to be generated from a dense specification of the computation, *versatility* leading to many different implementations of the same computation and *optimality* as an implementation optimal for a particular architecture can always be found in the different generated implementations. In this section, these characteristics are discussed in turn.

11.5.1 Applicability

The extended *forelem* framework that is proposed in this chapter is applicable to a large variety of existing codes. To demonstrate this, in this section we will show how a C code is translated to a tuple program. The C code that we will consider in this section is a simple triangular solver of lower triangular unit matrices:

```
for (int i = 0; i < N; i++)
  for (int j = i + 1; j < N; j++)
    B[j] = B[j] - A[j][i].value * B[i];
```

`B` is a vector consisting of `N` elements and `A` is an $\mathtt{N} \times \mathtt{N}$ matrix. First, the translation of data accessed by this loop into a tuple space is addressed. This translation is performed by forming tuples consisting of all array elements that are referenced by the statements in the loop. When a loop contains multiple statements, one tuple is formed containing all array elements accessed by all statements in the loop body. In fact, a bijection is set up between the iteration space of the original loop and the tuples.

In the case of the triangular solve example, the loop body only contains a single statement, resulting in the tuple:

$$(\mathtt{A[j][i]}, \mathtt{B[j]}, \mathtt{B[i]})$$

The fields of the tuple are labeled with `x`, `y` and `z`. The matrix and vector are stored as arrays of doubles. To store all values of `A`, $\mathtt{N}^2$ doubles are needed. This implies that $\mathtt{N}^2$ tuples are stored in the tuple space, so $\mathtt{N}^2$ doubles are needed to store all values for `B[j]` and another $\mathtt{N}^2$ for `B[i]`. The total necessary storage capacity is $3\mathtt{N}^2$ doubles.

The translation process tries to decrease the amount of storage that is necessary. Because two fields of the tuple are values from an array `B`, it replaces the values of `B` at this position with subscripts into `B`:

$$(\mathtt{A[j][i]}, \mathtt{j}, \mathtt{i})$$

Now, $\mathtt{N}^2$ doubles are needed to store the values `A`, $2\mathtt{N}^2$ integers to store the values for fields `y` and `z` and `N` doubles to store `B`. A total of $(\mathtt{N}^2 + \mathtt{N})$ doubles and $2\mathtt{N}^2$ integers. Considering doubles are typically stored in 8 bytes and integers in 4 bytes, the required storage capacity has almost been cut in half.

When the values for `A` in the tuples are replaced with subscripts, no storage saving space is achieved:

$$((\mathtt{j}, \mathtt{i}), \mathtt{j}, \mathtt{i})$$

To store these tuples, $4N^2$ integers are needed, N^2 doubles and N doubles. Since this does not present a saving of storage space over the previous tuple, the process selected the previous tuple as "loop tuple".

The loop bounds of the original loop indicate which of the loop tuples need to be stored in the tuple loop. In this case, the loop bounds are $i \in [0, N)$ and $j \in [i + 1, N)$. So, only tuples with y > z are to be stored in the tuple pool.

The next step in the translation process is to translate the loops to *forelem* loops, that operate on the tuple space defined by the selected loop tuple. This translation must preserve existing dependencies by translating these to corresponding *ready* clauses. The triangular solve loop that is being considered has a true dependency on array B: B[j] δ^t B[i]. This can be phrased as in order to execute the statement that reads from B[i], so z = i, all tuples that write to B[i], so y = i must have finished. So, the *ready* function is defined by the expression T[r].y == T[q].z. As a result, triangular solve can be expressed using a *whilelem* loop as follows:

```
forelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
  B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

11.5.2 Universality

All possible data structures can be represented in terms of tuples. This follows from the fact that computer memory in which these data structures are located can be represented in terms of tuples: e.g. (addr, value) tuples. In this subsection we demonstrate the universality of our approach by expressing linked list and tree travels as *whilelem* loops. Consider linked list links defined as:

```
struct List {
  void *data;
  struct List *next;
};
```

This linked list can be iterated in C with a regular *while* loop:

```
struct List *l = start;
while (l != NULL) {
  operate_on(l->data);
  l = l->next;
}
```

The body of this *while* loop accesses the values l, l->next and l->data, giving rise to tuples (x, y, z), containing the values (l, l->data, l->next). A *whilelem* loop can be written, which performs exactly the same iteration:

```
whilelem (l; l ∈ pT.ready(r)[T[r].z == T[l].x])
  operate_on(T[l].y);
```

The *ready* expression states that a tuple T[l].x can be visited, once all tuples T[r].z (z is the next field) have been visited. Note that this automatically leads to the first link of the linked list to be visited. No tuple exists with a next field

equal to the address of the first tuple in the list, therefore the empty set of tuples that precedes the first tuple in the list have all been visited.

Note that when a static execution schedule is generated for this *whilelem* loop and the data storage is materialized, the linked list is automatically linearized into an array. So, this process can be performed automatically with the generic tools provided within this framework, instead of with specific frameworks that have been developed for this in the past [94].

As another example, breadth-first traversal of a binary tree can be elegantly expressed as a *whilelem* loop. Let `T` be a tuple pool with tuples `(w, x, y, z)` corresponding to values `(n, n->data, n->left, n->right)` of a simple binary tree data structure. The following loop then visits the nodes of the tree in a breadth-first order:

```
whilelem (n; n ∈ pT.ready(r)[T[r].y == T[n].w ||
                             T[r].z == T[n].w])
  operate_on(T[n].x);
```

The ready condition specifies that a node `T[n]`, which is either the left or right child of a parent node, can be visited once the parent node has been visited. Because nodes are only marked as visited after execution of the *forelem* loop (which is embedded in the *whilelem* loop as discussed in Section 11.2.5), no new nodes become ready during the execution of the loop. As a consequence, there is the guarantee that nodes are indeed visited in a level-by-level order of the tree.

11.5.3 Transferability

Sparse matrix codes are often developed separately from dense matrix codes. This is because sparse matrices are stored in custom, pre-defined, data structures, contrary to dense matrix codes that store the data as a regular multi-dimensional array. It is this large difference in data structures that leads to the existence of separate code bases for dense and sparse matrix algebra.

Programs that are expressed in the extended *forelem* framework proposed in this chapter, can operate on both dense and sparse data storage, since both of these can be stored into a set of tuples that can be operated on. As a consequence, when a dense linear algebra computation is translated to a *forelem* loop, a sparse version of this routine can be derived automatically. To see this, consider tuples containing `(A[j][i], j, i)` from Section 11.5.1 and the loop body of the triangular solve code:

```
B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

As has been observed in the literature [15], this statement is a no-op in case `T[q].x == 0` and this implies that all tuples with `T[q].x == 0` can be omitted without affecting the end result of the computation. This corresponds with removing all matrix elements `A[j][i] == 0`. The result is an expression of the original computation that operates on sparse storage.

As another example, consider sparse matrix times vector multiplication:

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    C[i] = C[i] + A[i][j] * B[j];
```

this yields a tuple space T with tuples (i, A[i][j], j). Since this loop does not exhibit true dependencies, it can be expressed as a *forelem* loop without having to make use of a *ready* condition:

```
forelem (m; m ∈ pT)
  C[T[m].x] = C[T[m].x] + T[m].y * B[T[m].z];
```

Also in this case, tuples with T[m].y == 0 are no-op statements, and such tuples can be deleted from the tuple space without affecting the final result. Again, a transfer is made to a sparse data storage. By applying different transformations that are supplied with the *forelem* framework to this loop nest, different forms of this sparse data storage are automatically generated. For example, through Orthogonalization and Materialization the loop nest can be put into a form such that the computation is performed on a row-by-row basis and the tuples are explicitly organized in a row-by-row order in the array PT:

```
forall (i = 0; i < N; i++)
  forelem (m; m ∈ ℕ*)
    C[i] = C[i] + PT[i][m].y * B[PT[i][m].z];
```

where $\mathbb{N}* = [0, |\texttt{PT[i]}|)$. Further transformations may, for example, lead to a loop from which Compressed Row Storage is derived:

```
forall (i = 0; i < N; i++)
  forall (m = PT_ptr(i); m < PT_ptr(i+1); m++)
    C[i] = C[i] + PT[m].y * B[PT[m].z];
```

Other sparse data formats that can be generated automatically include, for example, Jagged Diagonal Storage, a sparse data storage format that up till now could not be derived automatically.

11.5.4 Versatility

The extended *forelem* framework that is proposed in this chapter, is backed by a versatile transformation framework that contains transformations that restructure the computation (projection, scheduling, orthogonalization), data storage (orthogonalization and concretization) as well as transformations that address efficiency of the final code (loop blocking, loop unrolling). Using these transformations, a search space is set up that contains many different variants of the same loop with different performance characteristics.

Chapter 10 showed that this search space contains at least 130 principal forms of the sparse matrix times k vector multiplication and 76 principal forms of the triangular solve computation. A principal form is an instantiation of the computation with a different loop structure and different data storage derived from this loop structure. In Figure 11.1 we plot again the transformation tree leading to the

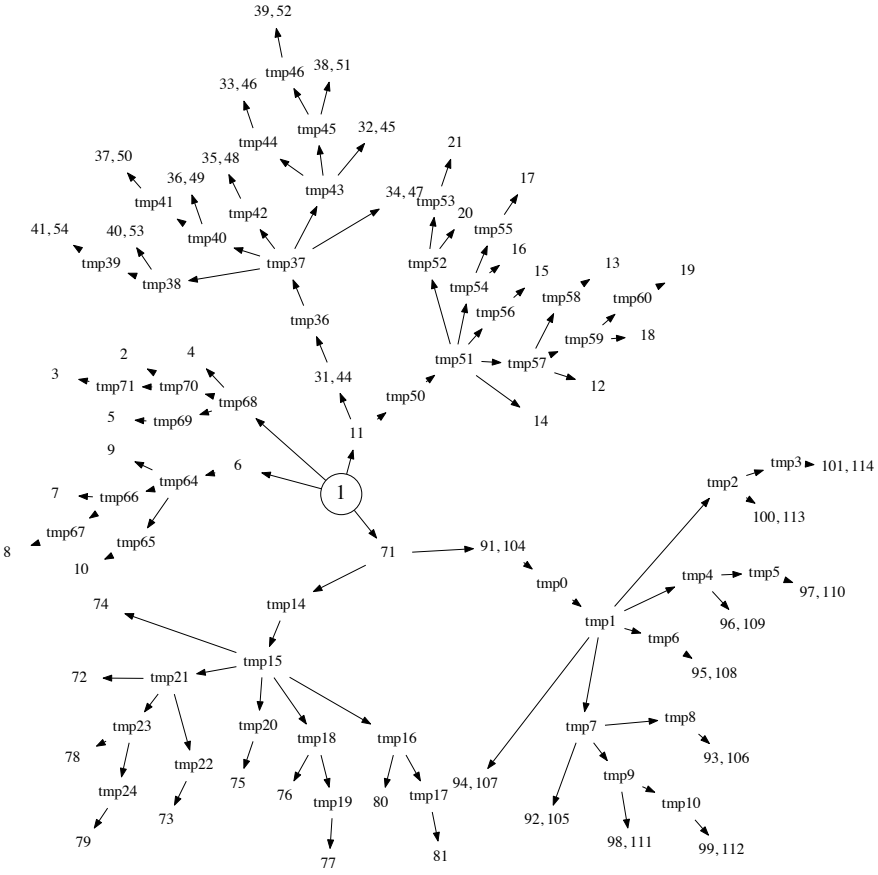


Figure 11.1: Transformation tree of the triangular solve computation. Through the application of transformations on the starting point *1, 76* different principal forms are generated.

76 different principal forms of triangular solve, see also page 178. The form in the center, labeled with *1*, is the starting point. The edges represent transformations that result in many different principal forms. The nodes that are prefixed with "tmp" are intermediate stages of the transformation process for which no executable is generated and these do not count as principal form.

11.5.5 Optimality

For all computations that are expressed in the extended *forelem* framework, either directly, or indirectly through a mapping from an original program code written in for instance C, a very large search space of possible implementations is set up. As we have described, the implementations in this search space have different orders in which the computation is carried out, different execution schedules leading to different parallelizations of the computation, different data structures and are subjected to different final parametrized optimizations that tune the executable code.

In this large search space, implementations of the computation can be found that are optimal for a given target architecture and make best use of the architecture's resources. The optimal implementation will vary for architecture, class of input data, etc. Note that because this search space is significantly larger than the search space that is exploited by contemporary compilers, optimal implementations can be found that are not found by contemporary compilers. The results of the work discussed in the previous subsection show that for the sparse matrix times k vector(s) multiplication always an implementation is found that is faster than the implementations supplied by several hand-optimized sparse algebra libraries, speedups are observed as large as 46%. Automatically generated implementations for triangular solve achieve in the majority of cases a speedup, up to 30% to 56% and in some cases no speedup is reported but the performance of the generated implementation is at least on par with the implementations provided by the sparse algebra libraries. For a further discussion of optimality, see also the next section.

11.6 Case Study: Triangular Solve

In this section, we demonstrate that from an ordinary triangular solve code in C, within the extended *forelem* framework parallelized implementations are produced, that up till now could only be derived by hand. The performance of these parallelized implementations is compared with that of a hand-optimized triangular solver.

11.6.1 Transformation Process to Produce Parallelized Implementations

The starting point is a triangular solve code written in C. This is a lower triangular solve code for unit matrices:

```
for (int i = 0; i < N; i++)
  for (int j = i + 1; j < N; j++)
    B[j] = B[j] - A[j][i] * B[i];
```

B is a vector consisting of N elements and A is an $N \times N$ matrix. As has been shown in Section 11.5.1, the optimization process will select (A[j][i], j, i) as an appropriate loop tuple for this loop. The fields of the tuples are named x, y, and z. Derived from the loop bounds, a tuple space is created that contains tuples for j > i, so tuples with fields such that y > z.

```
whilelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
  B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

Now that the computation is expressed within the extended *forelem* framework, transformations are applied and a mapping is done onto one of the execution models. From a different application of the transformations and different selections of execution models, many different implementations are generated. For example, in Section 11.3.1 a mapping of the triangular solve loop onto the affine embedding execution model was derived:

```
for (j; j ∈ f_y(T.y))
  for (i; i ∈ f_z(T.z))
    forelem (q; q ∈ pT.(y,z)[(j,i)])
      B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

From the facts that f_y and f_y are identity functions and that only tuples with y > z are present in the tuple pool, can be deduced that the inner *for* loop with iteration variable i can be executed in parallel. This loop is eliminated to give the *forelem* loop another degree of freedom:

```
for (j; j ∈ f_y(T.y))
  forelem (q; q ∈ pT.y[j])
    B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

This way, the 76 different principal forms as described in Section 11.5.4 can be generated from different affine embeddings. Additionally, further transformations will lead to a loop that operates on an efficient sparse data storage, such as described in Section 11.5.3.

Also implementations will be produced by mapping the computation onto the static execution model. For example:

```
l = 0;
reset (T);
while (unmarked?(T)) {
  forelem (q; q ∈ pT.ready(r)[T[r].y = T[q].z])
    T[q].level = l;
  l = l + 1;
}
```

```
for (l' = 0; l' < l; l'++)
  forelem (q; q ∈ pT.level[l'])
    B[T[q].y] = B[T[q].y] - T[q].x * B[T[q].z];
```

In this implementation, the ready clauses are first processed to find out which tuples can be processed at the same time. The loop nest that follows no longer needs the ready clause in order to perform the computation correctly, but rather uses the resulting "level" information. A static schedule is beneficial when the second loop nest, that actually performs the computation, is carried out multiple times for the same tuples, but a different array B.

This implementation is similar to implementations of triangular solvers described in the literature, that consist out of an processing step, or analysis phase, to analyze the structure of the sparse matrix and a solve phase that uses the result of this analysis to perform a highly parallel triangular solve computation [8, 72]. Contrary to the implementations described in the literature, which are derived by hand by an expert programmer, the implementation described in this section is produced automatically from an ordinary dense version of triangular solve using the extended *forelem* framework described in this chapter.

Further transformations can be performed. For example, Materialization, as described in Chapter 9, can be used to materialize the tuples into a two-dimensional array, where the tuples are stored in a level-by-level order. This results in the following computation loop nest:

```
for (l' = 0; l' < l; l'++)
  forelem (q; q ∈ ℕ*)
    B[PT[l'][q].y] = B[PT[l'][q].y] -
                     PT[l'][q].x * B[PT[l'][q].z];
```

where $\mathbb{N}* = [0, |\texttt{PT}[\texttt{l}']|)$.

11.6.2 Experimental Evaluation

We have conducted a preliminary experimental evaluation with codes that have been generated using the extended *forelem* framework. Various implementations have been generated for the affine embedding execution model and one implementation for the static scheduling execution model. CUDA codes have been generated that were compiled with the CUDA 5.0 toolkit. The resulting executables have been timed on a machine containing an Intel Xeon E5-2650 CPU at 2.00GHz, hosting an NVidia Telsa K20m GPU with 4799MB of RAM. The experiments were run for 16 matrices, obtained from the University of Florida Matrix Collection [28]. These matrices were reformatted to only store the lower triangle of the matrix and ones were placed on the diagonal, the sparsity patterns were preserved.

In Table 11.1, the results are shown for 8 different implementations based on the affine embedding execution model. All reported execution times are in milliseconds. These 8 implementations differ both in loop structure, as well as in data storage format. The data storage formats are also the result of transformations carried out by the compiler, that have modified the tuples into a form suitable for the computation to be carried out. As can be seen in the table, all implementations

	I.	II.	III.	IV.	V.	VI.	VII.	VIII.
Erdos971	3.04	1.73	2.93	1.64	2.38	0.919	2.34	0.904
mcfe	5.97	3.19	5.55	2.81	4.15	1.63	4.17	1.6
blckhole	13.9	8.12	13.5	7.75	10.8	4.32	11	4.31
OPF_10000	347	180	328	162	229	104	217	96
lhr71	1420	320	1370	273	361	162	351	148
stomach	1980	920	1870	837	1160	538	1150	501
3dtube	794	242	756	185	264	128	246	117
orsreg_1	14.2	8.46	13.7	8.08	11	4.59	11.3	4.58
shipsec1	1150	712	1010	562	780	372	740	346
shipsec5	1540	905	1360	718	1020	477	944	442
pdb1HYS	313	192	279	145	204	93.8	191	86.5
or2010	1270	792	1210	719	1020	442	1070	407
G2_circuit	946	607	908	575	771	341	830	332
144	1450	636	1350	568	794	371	745	345
cop20k_A	1450	526	1380	466	650	289	617	264
consph	708	432	658	332	473	221	452	205

Table 11.1: Execution time of different implementations based on the affine embedding execution model. All times reported are in milliseconds.

exhibit different performance characteristics. Some implementations are clearly faster than other implementations for all surveyed matrices. In this case, implementation *VIII.* always presents the fastest execution time.

With the extended *forelem* framework, one implementation was generated that is based on the static execution model. This implementation is especially suited when the same matrix is processed multiple times, with different values for the vector B. We have conducted a preliminary comparison of the performance of the automatically generated implementation to the performance of the parallelized triangular solve provided with the CUDA 5.0 toolkit in the CUSPARSE library [72].

The implementation supplied by the CUSPARSE library consists of both an analysis and solve phase that run on the GPU. The automatically generated implementation performs the analysis phase on the CPU and runs the computation loop nest on the GPU. For the *or2010* matrix, the automatically generated analysis phase, running on the CPU, is 20 times slower than the CUSPARSE analysis running on the GPU. Even though the execution times for the solve phase are the ones of prime interest since the analysis phase only has to be performed once per matrix, we intend to address this deficiency in future work.

The execution time of the solve phase of CUSPARSE and the automatically generated implementation using the extended *forelem* framework are shown for the 16 different matrices in Table 11.2. For a number of matrices, e.g. *stomach*, *shipsec1* and *G2_circuit*, the automatically generated implementation is faster than the implementation provided by CUSPARSE. This exemplifies the strength of our approach, due to the use of versatile transformations, implementations can be automatically found that beat hand-optimized implementations. In most other cases, the performance of the automatically generated implementation is with a difference of a factor of 2 to 3 competitive with the hand-optimized version. Notable

	CUSPARSE	Generated
Erdos971	0.136	0.193
mcfe	1.66	3.33
blckhole	0.456	1.19
OPF_10000	0.443	0.493
lhr71	0.881	8.03
stomach	22.1	16.5
3dtube	49.6	81.7
orsreg_1	0.279	0.461
shipsec1	24.3	15.6
shipsec5	32.0	20.4
pdb1HYS	100.0	128.0
or2010	0.86	0.937
G2_circuit	8.58	7.79
144	6.47	6.72
cop20k_A	2.93	2.83
consph	7.1	5.46

Table 11.2: Execution times of the solve phase of the triangular solve algorithm of both CUSPARSE and the automatically generated implementation using the extended *forelem* framework. Times reported are in milliseconds.

exception is the *lhr71* matrix, for which a approximately factor of 10 slowdown is observed.

11.7 Conclusions

In this chapter, we have described an extension to the *forelem* framework for the expression of data dependencies and the optimization of irregular parallel computations. By expressing computations using the ready clause, more information is exposed to the compiler about the order in which data is visited. Furthermore, the compiler is enabled to reorganize the processed data in a more optimal form, next to the ability to restructure the computation.

Another major benefit of this programming model is that from dependency information that is expressed as dependencies between tuples in a tuple pool, it is trivial to deduce which operations on tuples can be executed at the same time. As a consequence, this framework is especially suited for the automatic parallelization of irregular codes. We have described that through the application of transformations that are implemented in the underlying optimization framework, many different implementations can be generated with different performance characteristics and that an effective exploitation of this search space leads to automatically generated implementations that are competitive with hand-optimized codes.

We have shown that from an ordinary (dense) triangular solve code written in C, a highly parallelized implementation can be generated automatically, that computes on sparse data storage. Preliminary experiments that have been conducted, demonstrate that the performance of this automatically generated implementation is competitive to the performance of hand-optimized codes.

CHAPTER 12

Controlling Distributed Execution of Forelem Loops

12.1 Introduction

This chapter presents the results of a preliminary investigation into the suitability of the *forelem* framework for the automatic parallelization of database applications or automatic generation and optimization of Big Data applications for execution on multiple compute nodes. As has been discussed in this thesis, the *forelem* framework introduces a universal approach for the optimization of an application's data layout and storage. By incorporating details about the data access performed by the application into the optimization process, the application and its data access method can be synchronized. This synchronization leads to a better alignment of the application's computational loops with the order in which data is accessed. This can even lead to changes in the storage layout and format of the application. In this chapter extensions to the *forelem* framework are described so that data distribution can be explicitly controlled in addition to (local) data layout.

The approach that has been chosen for these extensions relies on the techniques derived from the optimization of program code and data distribution to map program codes onto parallel computers, see for instance [51]. In the *forelem* framework the distribution of data is being handled by special loop constructs which express parallel execution coupled with data decomposition. This data decomposition can be specified by an "automatic" decomposition of the value range of one or more particular fields in the database model. This approach is very generic allowing multiple data decompositions to be considered at compile time.

Big Data applications can also take advantage of the ability of the *forelem* framework to support vertical integration of application code and data access frameworks. Similar to regular database applications, Big Data applications typically access data through a framework that abstracts away peculiarities of accessing a particular file format, database system or distributed file system. Such frameworks inhibit optimizing compilers from potentially optimizing data access as

performed by an application. Through the use of the *forelem* framework, the data access operations that are performed through a data access framework are expressed in the generic intermediate representation, unlocking many more potential optimization opportunities.

This chapter takes the following approach: first a scheme is described to influence the distribution of *forelem* loops and the selection of a data distribution. Secondly, an important transformation in the context of distributed applications, Iteration Space Expansion, is described in detail. Thirdly, it is demonstrated how this scheme can be used together with the compiler optimization techniques that are implemented in the *forelem* framework to automatically parallelize database applications and generate Big Data codes. On a set of loops parallelized using a data distribution based on the value range of one of the fields, we demonstrate a number of transformations that optimize for the re-use of a selected data distribution. As a consequence, the *forelem* framework has the power to automatically distribute program codes similar to MapReduce-style computations. In other words, optimizing compiler technology is enabled to be used in the optimization and parallelization of database applications.

The viability of the *forelem* framework to be used for the optimization of Big Data applications is illustrated using two typical MapReduce examples. We show how from a problem expressed in SQL, a *forelem* intermediate representation of the problem can be derived and be subsequently optimized through the application of transformations present in the *forelem* framework. From this intermediate representation, different codes can be generated using different data layouts. Initial experiments show the importance of considering a good data layout for the problem at hand. Implementations generated with the *forelem* framework realize a performance improvement of a factor 3 compared to a Hadoop implementation when the same input data file is used. Performance improvements up to a factor 120 can be reached if the input data with an optimized layout is automatically generated.

This chapter is organized as follows Section 12.2 describes the extensions to the *forelem* framework for expressing distributed execution of *forelem* loops. Section 12.3 discusses the Iteration Space Expansion transformation. Section 12.4 illustrates how the proposed extension and the transformations in the *forelem* framework are used together and reinforce each other. Section 12.5 illustrates the viability of the *forelem* framework to optimize Big Data Applications. Section 12.6 discusses the conclusions.

12.2 Distribution of Forelem Loops

In Chapter 11 we used the observation that since *forelem* loops may iterate their index set in any order, *forelem* loops are inherently parallel. So, in fact, no special semantics are needed to be able to execute a *forelem* loop in parallel. However, when execution of a *forelem* loop is to be distributed to multiple nodes, control over what parts of the *forelem* loop is executed on what node is necessary to be able to optimize data decomposition and distribution. In the remainder of this

chapter, with "parallel execution", we mean parallel execution distributed over multiple processors (or compute nodes).

Within the *forelem* framework, parallelization consists out of loop scheduling, which is the problem of scheduling a parallel loop's iterations onto the available processors, and secondly out of data distribution (or decomposition) to the processors. It is assumed that accesses to data that is not available locally are resolved by performing remote communication to a processor that does have the necessary data available. Loop scheduling is implemented through the application of Loop Blocking to the iteration space of a *forelem* loop. A distinction is made between *direct* and *indirect* loop scheduling, both of which will be described in this section. Finally, it is shown how the data set can be decomposed based on the created loop schedule.

Loop scheduling follows from the application of Loop Blocking to the iteration space of a *forelem* loop. With *direct* loop scheduling, the iteration space is blocked by partitioning the index set that is iterated by the *forelem* loop. On the other hand, *indirect* loop scheduling is achieved by blocking on the value range of a field in the accessed array.

As an example, consider an array `A` with fields *field1* and *field2*, and the following loop where *SEQ* denotes a sequence of statements:

```
forelem (i; i ∈ pA)
  SEQ;
```

In order to parallelize this loop to N processors, a loop schedule must be created. To create a direct loop schedule, Loop Blocking splits the iteration space of this loop, which is the index set `pA`, into N partitions:

$$\mathtt{pA} = \mathtt{p}_1\mathtt{A} \cup \mathtt{p}_2\mathtt{A} \cup \ldots \cup \mathtt{p}_N\mathtt{A}$$

and the *forelem* loop becomes:

```
for (k = 1; k <= N; k++)
  forelem (i; i ∈ p_kA)
    SEQ;
```

Subsequently, to parallelize this loop to N processors, each processor must be assigned a partition of the index set `pA`. This is achieved by replacing the *for* loop with a *forall* loop, indicating that the outer loop is executed in parallel:

```
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ p_kA)
    SEQ;
```

As a next step, the data can be decomposed according to the selected partitioning. So, a decomposition of table `A` is created:

$$\mathtt{A} = \mathtt{A}_1 \cup \mathtt{A}_2 \cup \ldots \cup \mathtt{A}_N$$

based on the partitioned index sets $\mathtt{p}_k\mathtt{A}$. Note that, this decomposition of `A` yields an index set $\mathtt{pA}_k$ for every $\mathtt{A}_k$. The loop operating on the decomposed data is:

```
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ pA_k)
    SEQ;
```

where in the loop body data is accessed through for example $\texttt{A}_k[\texttt{i}].\texttt{field1}$. Note that, in case data accesses are performed to data that is not available locally after the data decomposition, these accesses can be resolved by performing remote communication to a processor that does have the necessary data available.

In indirect data partitioning, Loop Blocking is not done based on the iterated index set, but on the value range of one of the table's accessed fields. Consider the same starting point:

```
forelem (i; i ∈ pA)
  SEQ;
```

Array A is to be distributed into N partitions based on *field1*. The notation *A.field1* denotes the set of values of the *field1* found in all subscripts of A. If X = A.field1, then

$$\texttt{X} = \texttt{X}_1 \cup \texttt{X}_2 \cup \ldots \cup \texttt{X}_N$$

is a partitioning of X into N segments. The blocked loop is:

```
for (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA.field1[l])
      SEQ;
```

In this loop nest the outer loop can be parallelized. In the parallelized loop nest a processor P_k is responsible for processing partition $\texttt{X}_k$ of this partitioning and will execute the original *forelem* loop only for $i \in \texttt{pA}, l \in \texttt{X}_k : \texttt{A}[\texttt{i}].\texttt{field1} = l$. This results in:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA.field1[l])
      SEQ;
```

Also in this case, the table A can be decomposed based on the selected *indirect* loop schedule. The decomposition of A into N parts $\texttt{A}_k$, with corresponding index sets $\texttt{pA}_k$ is based on the partitioning X into $\texttt{X}_k$. This results in the following loop nest:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA_k.field1[l])
      SEQ;
```

where the loop body accesses, for example, $\texttt{A}_k[\texttt{i}].\texttt{field1}$. Note that, this data decomposition guarantees that $\texttt{pA}_k$ only contains subscripts i such that values $\texttt{A}_k[\texttt{i}].\texttt{field1}$ are always contained in $\texttt{X}_k$. Based on this observation, the loop can be simplified to:

```
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ pA_k)
    SEQ;
```

without affecting the final result.

Within the *forelem* framework, the optimization of the data distribution is performed after the selection and optimization of the data partitioning or loop scheduling. The process of data distribution optimization depends on the communication model that is used to transfer data between processors, on any initial data distribution that is present and on the loop schedules that have been selected for other *forelem* loops in the application that access the same data.

Many static and dynamic approaches to loop scheduling have been described in the literature [79, 92, 19]. A static loop schedule is determined entirely at compile-time. Dynamic approaches schedule iterations to idle processors at runtime and have the opportunity to better balance the load in case the cost for each loop iteration is not equal.

An example dynamic scheduling approach is Guided Self-Scheduling (GSS) [79]. In GSS, iterations of loops are scheduled to idle processors at runtime. Iterations are allocated in groups called chunks. The process starts with a large chunk size and this size gradually decreases with the course of execution. The next chunk size to use is determined by dividing the number of remaining iterations by the number of processors. Processors that finish their chunk earlier than other processors are assigned a new smaller chunk. This technique results in a better balancing of the work.

12.3 Iteration Space Expansion

This section introduces the Iteration Space Expansion transformation, which is an important transformation to enable efficient distributed execution of *forelem* loops. Furthermore, the transformation can turn turn *forelem* loops into a form suitable for MapReduce-like processing. Before this transformation is described in detail, a number of related transformations are reviewed.

Previous in Chapter 6 Iteration Space Expansion has been described. Iteration Space Expansion is advantageous in codes that exhibit irregular access patterns that are made regular by iterating the (expanded) iteration space in which the irregular accesses are contained. For example, consider the following loop from a sparse matrix code [95]:

```
for (i = 0; i < N; i++)
  {
    for (q ∈ colIndex(Δ))
      {
        result[i] += M'[i, q] * right[q];
      }
  }

Δ = {start[i], start[i] + 1, ..., start[i + 1] - 1}
```

where colIndex exhibits an irregular access pattern. The iteration space of the inner loop is expanded to iterate the entire positive integer range:

```
for (i = 0; i < N; i++)
  {
    for (q = 0; q < INT_MAX; q++)
      {
        result[i] += M''[i, q] * right[q];
      }
  }
```

where M''[i, q] is defined by

```
if (q ∈ colIndex(Δ))
  M''[i, q] = M'[i, q];
else
  M''[i, q] = 0;
```

so that the semantics are preserved, because for subscripts i, q for which M' is not defined, 0 is returned. This code is an intermediate step in an optimization process and enables new optimization opportunities because the two loops are now regular.

For Iteration Space Expansion in the context of *forelem* loops, the following example is considered:

```
count = 0;
forelem (i; i ∈ pA.field[X])
  count++;
tmp = count;
```

Essentially, the example counts the number of array subscripts for which field equals a value X. This can be seen as the computation of an aggregate value, for example as part of a group-by computation where the aggregate is computed for different values X. The example can be rewritten as follows, with the condition made explicit:

```
count = 0;
forelem (i; i ∈ pA)
  if (X == A[i].field)
    count++;
tmp = count;
```

Let us recap Iteration Space Expansion as described in Chapter 6. The Iteration Space Expansion transformations now consists of three steps. Firstly, the condition on *field* is eliminated, so that the body of the *if* statement (the actual loop body) is executed for all subscripts of A. In fact, the iteration space is expanded from pA.field[X] to pA. Secondly, the scalar count is expanded to a vector, subscripted by A[i].field. Thirdly, any reference to the scalar count is rewritten to access the vector, with X as subscript. The value that is assigned to tmp is then equivalent to the value assigned in the original code. This results in:

```
count[] = 0;
forelem (i; i ∈ pA)
  count[A[i].field]++;
tmp = count[X];
```

The transformation is generalized as follows, see also Chapter 6. For a loop of the form

```
forelem (i; i ∈ pA.field[X])
  SEQ;
```

the following steps are performed:

1. the condition `A[i].field == X` is removed, which expands the iteration space so that the entire array `A` is visited,

2. scalar expansion is applied on all variables that are written to in the loop body denoted by *SEQ* and references to these variables are subscripted with the value tested in the condition, in this case `A[i].field`,

3. all references to the scalar expanded variables after the loop are rewritten to reference subscript `X` of the scalar expanded variable.

As an additional example of the transformation, consider the following loop which computes the average of a set of values:

```
count = 0;
sum = 0;
forelem (i; i ∈ pA.field1[X])
  {
    sum += A[i].field2;
    count++;
  }
tmp = sum / count;
```

When the same transformation steps are carried out, the result is:

```
count[] = 0;
sum[] = 0;
forelem (i; i ∈ pA)
  {
    sum[A[i].field1] += A[i].field2;
    count[A[i].field1]++;
  }
tmp = sum[X] / count[X];
```

A useful application of the Iteration Space Expansion transformation is in loop nests that compute an aggregate function for a series of values. For example:

```
forelem (i; i ∈ pA.distinct(field1))
  {
    count = 0;
    forelem (j; j ∈ pA.field[A[i].field])
      count++;
    ℛ = ℛ ∪ (A[i].field, count)
  }
```

which computes the count aggregate function for all distinct values of field1 in array A. The Iteration Space Expansion transformation is applied to the inner loop, to result in:

```
forelem (i; i ∈ pA.distinct(field1))
  {
    count[] = 0;
    forelem (j; j ∈ pA)
      count[A[j].field]++;
    ℛ = ℛ ∪ (A[i].field, count[A[i].field])
  }
```

The inner loop that computes the count array is now fully independent of the outer loop. Loop Invariant Code Motion is applied to move the inner loop out of the outer loop:

```
count[] = 0;
forelem (j; j ∈ pA)
  count[A[j].field]++;
forelem (i; i ∈ pA.distinct(field1))
  ℛ = ℛ ∪ (A[i].field, count[A[i].field])
```

As a result of the preceding transformation, the array A only has to be iterated once to compute all aggregates, at a cost of higher memory usage to store the count array. In fact, the loop resulting from this chain of transformations is similar to a hash aggregation strategy that is used in database systems. Furthermore, the first loop allows for straightforward parallelization:

```
count[] = 0;
forall (k = 1; k <= N; k++)
  forelem (j; j ∈ p_kA)
    count[A[j].field]++;
forelem (i; i ∈ pA.distinct(field1))
  ℛ = ℛ ∪ (A[i].field, count[A[i].field])
```

12.4 Illustration of the application of transformations

This section illustrates how the distribution of *forelem* loops over multiple processors described in Section 12.2, the transformations defined in the *forelem* framework (see for instance Chapter 3), and the Iteration Space Expansion transformation described in the previous section are used together and reinforce each other.

In particular, we will show that two adjacent loops which access the same table and are distributed based on *different* fields of this table can be transformed so that both loops use the same data distribution and a costly data redistribution is not necessary.

As a starting point, the following two adjacent loops on *Table* are considered:

```
forelem (i; i ∈ pTable)
  SEQ;
...
forelem (i; i ∈ pTable)
  SEQ;
```

where the first loop is distributed based on *field1* and the second loop on *field2*:

```
forall (j = 1; j <= N; j++)
  for (k ∈ X_j)
    forelem (i; i ∈ pTable.field1[k])
      SEQ;
...
forall (j = 1; j <= N; j++)
  for (k ∈ X_j)
    forelem (i; i ∈ pTable.field2[k])
      SEQ;
```

Even if `Table.field1` $\equiv$ `Table.field2` and the two decompositions are the same, data partitioning conflicts can occur. This is because a partitioning of *Table* based on *field1* is not equal to a partitioning of *Table* on *field2*. The fact that the column contents are equal does not imply the column contents are in the same order (the columns are multisets).

To resolve this, either *Table* is not distributed for the first loop or a redistribution of the table data is performed in between the first and second loop. Evidently, both are suboptimal solutions. However, if the *forelem* loop bodies compute an aggregate function, then a solution is possible.

For instance, assume `SEQ` of the first loop consists of incrementing a counter, thereby computing the multiplicity of all values of *field1* in *Table*. An outer loop is required to perform the aggregate function for every distinct *field1* value in *Table*, and the first loop results in:

```
forelem (i; i ∈ pTable.distinct(field1))
{
  count = 0
  forelem (j; j ∈ pTable.field1[Table[i].field1])
    count++;
  R_1 = R_1 ∪ (Table[i].field1, count)
}
```

This loop nest is suboptimal as it makes multiple passes through *Table* in the inner loop. To enable parallelization of this loop nest, Iteration Space Expansion is used, resulting in:

```
forelem (i; i ∈ pTable.distinct(field1))
{
  // Initialize count to zero for all dimensions
  count[] = 0
  forelem (j; j ∈ pTable)
    count[Table[j].field1]++;
  ℛ1 = ℛ1 ∪ (Table[i].field1, count[Table[i].field1])
}
```

Because the computation of count is now independent of i, the loop computing the count array can be moved out of the enclosing loop:

```
count[] = 0
forelem (j; j ∈ pTable)
  count[Table[j].field1]++;
forelem (i; i ∈ pTable.distinct(field1))
  ℛ1 = ℛ1 ∪ (Table[i].field1, count[Table[i].field1])
```

As a next step, the loop computing the count array is parallelized:

```
count[] = 0
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (j; j ∈ pTable.field1[l])
      count[Table[j].field1]++;
forelem (i; i ∈ pTable.distinct(field1))
  ℛ1 = ℛ1 ∪ (Table[i].field1, count[Table[i].field1])
```

Writes to count are performed to a global array which potentially generates significant amount of communication. (However, careful analysis will indicate that the writes to count are in this case controlled by the distribution of X, such that no two distinct nodes will write to the same subscript of count). The amount of communication can be reduced by creating a local array $\texttt{count}_k$ for each processor P_k:

```
count[] = 0
forall (k = 1; k <= N; k++)
{
  count_k = 0
  for (l ∈ X_k)
    forelem (j; j ∈ pTable.field1[l])
      count_k[Table[j].field1]++;
}
forelem (i; i ∈ pTable.distinct(field1))
{
  count[Table[i].field1] = Σ_{k=1}^{N} count_k[Table[i].field1]
  ℛ1 = ℛ1 ∪ (Table[i].field1, count[Table[i].field1])
}
```

An alternative parallelization approach is to parallelize both loops and not only the loop computing the `count` array. As a consequence of parallelizing the loop creating the result table, the processors should create partial result tables $\mathscr{R}_{1,k}$ which are later combined. Both loops are parallelized with `X = Table.field1`, which results in:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
  {
    forelem (i; i ∈ pTable.field1[l])
      count_k[Table[i].field1]++
    forelem (i; i ∈ pTable.distinct(field1))
      R_{1,k} = R_{1,k} ∪ (Table[i].field1, count_k[Table[i].field1])
  }
sum[] = 0
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ pR_{1,k})
    sum[R_{1,k}[i].field1] += R_{1,k}[i].count
forelem (i; i ∈ pTable.distinct(field1))
  R_1 = R_1 ∪ (Table[i].field1, sum[Table[i].field1])
```

Now, we return to the initial example. We consider two of the above loops, where the former is parallelized with `X = Table.field1` and the latter with `X = Table.field2`. Then the first two *forall* loops are:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
  {
    forelem (i; i ∈ pTable.field1[l])
      count_{1,k}[Table[i].field1]++
    forelem (i; i ∈ pTable.distinct(field1))
      R_{1,k} = R_{1,k} ∪ (Table[i].field1, count_{1,k}[Table[i].field1])
  }
...
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
  {
    forelem (i; i ∈ pTable.field2[l])
      count_{2,k}[Table[i].field2]++
    forelem (i; i ∈ pTable.distinct(field2))
      R_{2,k} = R_{2,k} ∪ (Table[i].field2, count_{2,k}[Table[i].field2])
  }
```

As has been indicated at the beginning of this section, data partitioning conflicts will occur for these two loops. These could be solved by performing an expensive data redistribution in between the execution of these two loops. However, in this case a better solution is to exploit the possibility to reorder the loops such that the two parallelized loops computing the *count* aggregate are consecutive to one another. This is possible because these loops do not have a dependency on the other

loops (the second *forall* loops) in the code fragment. The two outermost loops iterate the same bounds, allowing application of the Loop Fusion transformation:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
  {
    forelem (i; i ∈ pTable.field1[l])
      count_{1,k}[Table[i].field1]++
    forelem (i; i ∈ pTable.distinct(field1))
      R_{1,k} = R_{1,k} ∪ (Table[i].field1, count_{1,k}[Table[i].field1])
    forelem (i; i ∈ pTable.field2[l])
      count_{2,k}[Table[i].field2]++
    forelem (i; i ∈ pTable.distinct(field2)))
      R_{2,k} = R_{2,k} ∪ (Table[i].field2, count_{2,k}[Table[i].field2])
  }
```

In the case that `Table.field1` $\equiv$ `Table.field2`, another series of statement reordering and Loop Fusion is possible in the loop body resulting in:

```
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
  {
    forelem (i; i ∈ pTable.field1[l])
    {
      count_{1,k}[Table[i].field1]++
      count_{2,k}[Table[i].field2]++
    }
    forelem (i; i ∈ pTable.distinct(field1))
      R_{1,k} = R_{1,k} ∪ (Table[i].field1, count_{1,k}[Table[i].field1])
    forelem (i; i ∈ pTable.distinct(field2))
      R_{2,k} = R_{2,k} ∪ (Table[i].field2, count_{2,k}[Table[i].field2])
  }
```

Because the two counting loops us the same partitioning of `X`, it is possible to fuse these two loops. In other words, the loops use the same data distribution and no data redistribution is necessary in between loops. This technique can be extended to other combinations of loops, such as for example:

```
forelem (i; i ∈ pTable)
  R = R ∪ (...)
...
forelem (i; i ∈ pR)
  SEQ;
```

The second loop consumes tuples produced by the first loop. If the second loop does not have any restricting dependencies, the body of the second loop can be moved to the position in the first loop where the tuples are produced. As a result, also in this case both loops make use of the same data distribution of *Table*.

Although the interaction of the different transformations is rather powerful, it should be noted that we have only considered one particular case of two consecutive *forelem* loops. In general, database applications are not that simple and consist of many queries, embedded or not embedded in application code, so, the complexity of these interactions will grow exponentially. Although not addressed in this thesis, it is important to reckon that strategies will have to be developed to keep the optimization process manageable.

12.5 Application on Big Data Programs

In this section, it will be illustrated how the described *forelem* framework is used to optimize Big Data applications. The two examples from the original MapReduce [30] paper are considered which process data that is typically acquired from the use of the World Wide Web: web server page request logs and a database of links between web pages. We show that, using the *forelem* framework and starting with a SQL representation of the problem, a MapReduce-like program can be automatically derived and that transformations can be applied as usual on the *forelem* representation of the problem. Secondly, to show the importance of a good data layout, we explore the performance of different codes generated for these examples, using different data layouts, and compare this performance to the implementation of the examples in Hadoop.

12.5.1 URL Access Count

The first example concerns URL access count. Consider logs of web page requests, which are mapped to tuples $(url, 1)$. The reduction operator is described in the paper as mapping $(url, list(values))$ to $(url, total\_count)$. Considering a table *access*, with a single column containing the URLs, this computation can be described as the following SQL query:

```
SELECT url, COUNT(url) FROM access GROUP BY url
```

The *forelem* framework will generate the following loop nests from this query:

```
forelem (i; i ∈ pAccess.distinct(url))
  𝒢 = 𝒢 ∪ (Access[i].url)
forelem (i; i ∈ p𝒢)
  {
    count = 0;
    forelem (j; j ∈ pAccess.url[𝒢[i].url])
      count++;
    ℛ = ℛ ∪ (𝒢[i].url, count)
  }
```

The inner loop of the *forelem* loop iterating p$\mathscr{G}$ suits the application of the Iteration Space Expansion transformation:

```
forelem (i; i ∈ pAccess.distinct(url))
  𝒢 = 𝒢 ∪ (Access[i].url)
forelem (i; i ∈ p𝒢)
  {
    count[] = 0;
    forelem (j; j ∈ pAccess)
      count[Access[j].url]++;
    ℛ = ℛ ∪ (𝒢[i].url, count[𝒢[i].url])
  }
```

As the inner loop has been made fully independent of the enclosing loop, the loop can be moved outwards:

```
forelem (i; i ∈ pAccess.distinct(url))
  𝒢 = 𝒢 ∪ (Access[i].url)
count[] = 0;
forelem (j; j ∈ pAccess)
  count[Access[j].url]++;
forelem (i; i ∈ p𝒢)
  ℛ = ℛ ∪ (𝒢[i].url, count[𝒢[i].url])
```

The transformation sequence so far has turned the multiple irregular accesses to the `Access` array into a single regular iteration. As a next step, the first and third loops are merged using a transformation called Table Propagation, which will propagate the loop creating table $\mathscr{G}$ to the loop accessing $\mathscr{G}$, effectively eliminating the streaming of a table between two loops:

```
count[] = 0;
forelem (j; j ∈ pAccess)
  count[Access[j].url]++;
forelem (i; i ∈ pAccess.distinct(url))
  ℛ = ℛ ∪ (Access[i].url, count[Access[i].url])
```

The first loop is a good candidate for parallelization. The index set `pAccess` is divided over N processors (*direct* loop scheduling) such that the loop can be executed in parallel:

```
count[] = 0;
forall (k = 1; k <= N; k++)
  forelem (j; j ∈ p_kAccess)
    count[Access[j].url]++;
forelem (i; i ∈ pAccess.distinct(url))
  ℛ = ℛ ∪ (Access[i].url, count[Access[i].url])
```

A problem with this loop nest is that all processors write to the shared `count` array to store the results, which will be a big bottleneck in the code that will be generated from this intermediate representation. To alleviate this, every processor is given a local `count` array, from which a single value is reduced in the second loop:

```
count_{1...N}[] = 0;
forall (k = 1; k <= N; k++)
  forelem (j; j ∈ p_kAccess)
    count_k[Access[j].url]++;
forelem (i; i ∈ pAccess.distinct(url))
  R = R ∪ (Access[i].url, Σ_{k=1}^{N} count_k[Access[i].url])
```

Note that this code fragment bears similarity to a MapReduce program. In fact, the first loop maps every row of *access* to an accumulation of the `Access[i].url` subscript of the `count` array. This could be represented as a tuple $(url, 1)$. The second loop iterates over all keys, which are all distinct URLs in *access* and retrieves the result of an aggregate function, in this case `count`.

In general, two adjacent *forelem* loops where the former loop stores values in an array subscripted by a field of the array being iterated, and the latter loop accesses elements of this array, can be written as a MapReduce program. The *map* function iterates the table that is iterated by the former loop. This table is fragmented by a MapReduce framework, so that each instance of the *map* function processes a table fragment. This corresponds with a data distribution for the above code fragment where each processor has the rows that are referenced by the index set fragment stored locally. Instead of writing to a global array, `emitIntermediate` is called. For the above example, tuples `(Access[i].url, 1)` are generated, where the 1 is a dummy value, because it is not used.

The example code increments the value stored in the `count` array for every occurrence of a value `Access[i].url`. In the MapReduce program, a pair will be generated for every `Access[i].url`. So, the reduction function has to increment a counter for every occurrence of the same value `Access[i].url`. Because a MapReduce framework will collect all pairs for a unique key, the reduction function simply needs to count all values for every unique key. If the above example is written in MapReduce pseudocode similar to that used in [30], the program would be:

```
map(key, value):
  # Assume value represents the content of the
  # access table
  access = value
  for a in access:
    emitIntermediate(a.url, 1)

reduce(key, values):
  count = 0
  for v in values:
    count++
  emit(key, count)
```

12.5.2 Reverse Web-Link Graph

As a second example from the MapReduce paper we consider the Reverse Web-Link Graph. For each link from a source to a target page, a pair $(target, source)$

is emitted. The original example reduces to a pair $(target, list(source))$, which we will modify to reduce to a pair $(target, source\_count)$. To write a SQL query for this program, consider a table *links* that contains tuples $(source, target)$, which has been previously filled, for example by parsing webpages *source* and extracting all links to target pages. The following two queries are defined:

```
CREATE VIEW target_links AS
  SELECT DISTINCT target FROM links;
SELECT T.target,
  (SELECT COUNT(*) FROM links L
   WHERE L.target=T.target)
FROM target_links T;
```

which compute the number of incoming links to each registered target page. Expression of these queries in the *forelem* framework results in the following loops:

```
forelem (i; i ∈ pLinks.distinct(target))
  𝒯 = 𝒯 ∪ (links[i].target)
forelem (i; i ∈ p𝒯)
  {
    count = 0;
    forelem (j; j ∈ pLinks.target[links[i].target])
      count++;
    ℛ = ℛ ∪ (links[i].target, count)
  }
```

Using Table Propagation (see Section 6.2.3, this can be turned into a single loop nest:

```
forelem (i; i ∈ pLinks.distinct(target))
  {
    count = 0;
    forelem (j; j ∈ pLinks.target[links[i].target])
      count++;
    ℛ = ℛ ∪ (links[i].target, count)
  }
```

Let us consider a different transformation chain for this example. The outer loop iterates all distinct values of *target*. In fact, the value range of `Links.target` is iterated. Let `X = Links.target` and parallelize the loop using indirect loop scheduling:

```
forall (k = 1; k <= N; k++)
  {
    for (l ∈ X_k)
      {
        count = 0;
        forelem (j; j ∈ pLinks.target[l])
          count++;
```

```
      ℛ = ℛ ∪ (l, count)
    }
  }
```

Subsequently, Iteration Space Expansion is applied on the inner loop and the loop is moved outwards one level:

```
forall (k = 1; k <= N; k++)
  {
    count[] = 0;
    forelem (j; j ∈ pLinks)
      count[links[j].target]++;
    for (l ∈ X_k)
      ℛ = ℛ ∪ (l, count[l])
  }
```

In the current loop, every processor will compute its own copy of the `count` array. Also, the processors will contend for access to the result table $\mathscr{R}$. One possibility is to give every processor a private copy of $\mathscr{R}$ and merge the copies to a final result table in the master node:

```
forall (k = 1; k <= N; k++)
  {
    count[] = 0;
    forelem (j; j ∈ pLinks)
      count[links[j].target]++;
    for (l ∈ X_k)
      ℛ_k = ℛ_k ∪ (l, count[l])
  }
ℛ = ⋃_{k=1}^{N} ℛ_k
```

Another possibility is to move the loop computing the array further outwards:

```
count[] = 0;
forelem (j; j ∈ pLinks)
  count[links[j].target]++;
forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    ℛ = ℛ ∪ (l, count[l])
```

And to undo the parallelization of the second loop:

```
count[] = 0;
forelem (j; j ∈ pLinks)
  count[links[j].target]++;
forelem (i; i ∈ pLinks.distinct(target))
  ℛ = ℛ ∪ (links[i].target, count[links[i].target])
```

Instead of parallelizing the second loop, the first loop can be selected for parallelization. This will result in loops similar in structure to the first example (see Section 12.5.1).

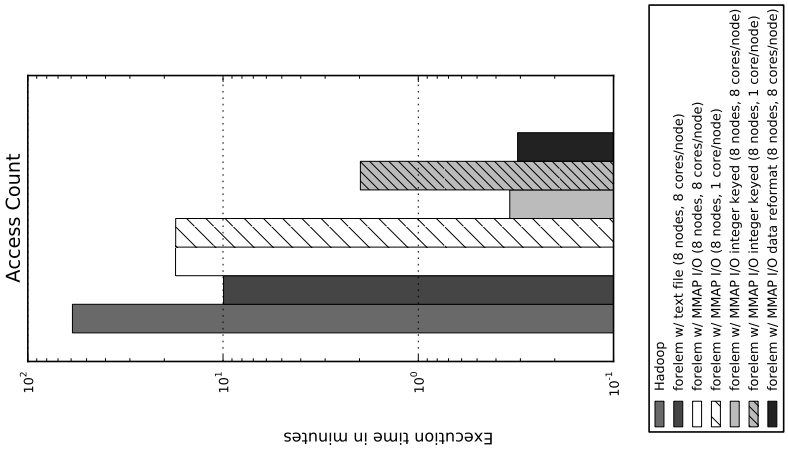


Figure 12.1: Execution time in minutes for the Hadoop implementation and various *forelem* implementations of the Access Count example.

12.5.3 Initial Performance Comparison

A number of initial experiments have been conducted with Hadoop and *forelem*-generated implementations of the two described examples. Different implementations were generated with the *forelem* framework, using different data layouts. The experiments have been performed on the DAS-4 cluster at Leiden University [48]. The cluster nodes each contain 8 processing cores, 48GB of main memory and 10 TB of local storage in a software RAID0 configuration. The Hadoop experiments were performed on a Hadoop cluster of 7 data nodes and one master node running the task tracker. The *forelem* implementation is a C code generated using the *forelem* framework, which uses MPI and OpenMP message exchange and local parallelization. This implementation is also run on 7 nodes and one separate master node.

The Access Count example has been run on a generated data file of 320GB, which is a comma separated file containing URL, data, server name that processed the request and a status code. The file has been stored onto the HDFS for processing with Hadoop and was evenly distributed over nodes according to a static, direct, loop schedule for processing by the *forelem* implementation. The Reverse Link count example has been run on a comma separated file containing source and target URL pairs. This file had a size of 177GB.

The results of these experiments are visualized in Figures 12.1 and 12.2. The numbers shown are averages of 4 runs, the variance between the experiments is negligible. The experiments show that the *forelem* implementations realize a performance improvement of a factor 3 when the same input data is used as is used by Hadoop, and up to a factor 120 if the input data is available with an optimized layout. It should be noted that if it is possible to reformat the data, large performance improvements can be achieved.

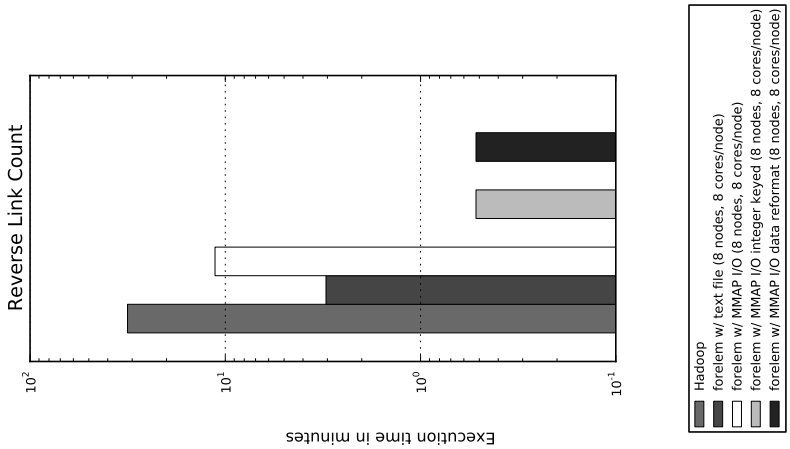


Figure 12.2: Execution time in minutes for the Hadoop implementation and various *forelem* implementations of the Reverse Link Count example.

Different versions of the *forelem* implementation have been generated. The first reads the text file in the same format as the Hadoop implementation. The code that has been generated from the *forelem* loop does not iterate through a pre-formatted array, but instead iterates the lines in the text file, which are split into the separate fields.

In the other experiments the use of a binary file that contains the pre-formatted array has been studied. This array is mapped into memory and processed by the code generated from the *forelem* intermediate representation. The experiments show that the use of such a format is not beneficial if the data file contains strings, due to the padding required in the binary format. As a result, the binary files are considerably larger than the text files and time taken by the I/O subsystem to read this file does not weigh up to the savings in parsing. When parallelism is reduced to a single core per node (i.e. to minimize the amount of disk seeks triggered by multiple processes reading different data from the disk) the performance does not improve. Instead, the CPU was kept busy, indicating that the disk read speed is not a problem when only a single thread per node is used. It is likely that there is a sweet spot where the number of threads is in balance with the throughput from the I/O system.

The *forelem* framework is capable of automatically reformatting the data layout of a program. As an example, the strings (URLs and hosts) in the arrays have been replaced with integer keys. These integer keys are used to subscript another array, which contains the string value for each key. In fact, the data model has been made relational. This significantly improves the performance, as indicated by the "integer keyed" experiments, which implies that it is worthwhile to consider such data reformatting if this is feasible in the context of the problem, for example when the data has not yet been collected in a specific format. A final experiment has been done by removing unused structure fields and column-wise storage of

the data. These data relayout operations can also be done automatically by the *forelem* framework. A performance increase is not observed after performing this relayout, possibly because it does not weigh up to the initial start up cost of the MPI and OpenMP frameworks.

12.6 Conclusions

This chapter described an extension of the *forelem* to express distributed execution of *forelem* loops. These extensions enable the *forelem* framework to exert control over the data distribution and decomposition across multiple compute nodes in addition to the control of (local) data layout as was discussed in Chapter 9.

In the context of distributed *forelem* codes, the Iteration Space Expansion transformation plays an important role. Through the use of this transformation and the other transformations present in the *forelem* framework, changes are not only made to the loop structure of the program but also to the data layout by reformatting this layout. Also, it has been described how these extensions can be used to translate a problem expressed in SQL to a parallelized *forelem* representation of the problem, from which a MapReduce-like program can be deduced.

The viability to use the *forelem* framework to optimize Big Data applications has been illustrated using two example MapReduce problems. These problems have been expressed in the *forelem* intermediate representation and were subsequently optimized. From this *forelem* intermediate representation different codes have been generated, using different data layouts. The performance of these different codes have been compared to a Hadoop implementation of the same problem. From these initial experiments follows that data layout plays an important role. When the same data file is used, performance improvements were obtained of at least a factor 3. If it is possible to reformat the data, the implementations generated using the *forelem* framework with reformatted data show performance improvements up to a factor 120.

CHAPTER 13

Summary & Future Perspectives

In this thesis, we have investigated a solution for the unification of imperative and declarative codes. This has resulted in the *forelem* intermediate representation. Declarative codes can be represented in the form of loops performing tuple accesses with simple loop control. These loops are especially suited to be made part of the workflow of traditional optimizing compilers. In fact, we have shown that many established compiler transformations can be re-targeted to operate on *forelem* loops. Through the application of these transformations, queries expressed in *forelem* loops can be optimized to performance comparable with that of contemporary state-of-the-art database systems (as described in Chapter 6) and transformations can be carried out that intertwine execution of the application code with the execution of the data access code as was illustrated by the examples in Chapters 3, 4 and 5.

The automatic reduction methodology discussed in Chapter 7 exemplifies the strength of the design of the *forelem* framework. The described prototype compiler, powered by the *forelem* framework, is capable of eliminating up to 90% of the instructions executed by two web applications. This process reduces database applications to their essence and unlocks more possibilities for the optimization of the performance of these applications.

Part of the automatic reduction process is to create a local copy of the data that is operated on. For large, distributed, deployments of web applications this may form a bottleneck. Whether this is the case depends on the application. In the case of e-business processing, the workloads are often read-dominant [64], so a local data copy is advantageous. On the other hand, when write actions are performed that trigger updates in all local copies of the data, the performance advantage of the vertically integrated application is most likely lost. To provide a solution to these problems, an analysis method is needed to determine when vertical integration and performing all reads on a local copy of the data is beneficial and when it is not. In Chapter 8 an initial study towards such an analysis was presented. Given a set of query mixes executed by an application it can be determined whether vertical integration will be beneficial.

The application of the *forelem* framework in the domain of database applications has been successful. Due to the generic nature of this framework, we found that this framework is also applicable to other application domains. In Part II of this thesis the application of this framework in a number of these different application domains has been studied. A focus was the optimization of the used data storage format together with the code operating on this data. Data structures are in fact reassembled, by first translating the data structure into tuples on which a sequence transformations are performed leading to the generation of a new data storage format. Transformations that affect the generation of the final data storage format were described in Chapter 9.

These transformations set up a large search space of possible loops and data storage formats that can be generated from a single initial *forelem* loop. In Chapter 10 this search space was characterized and explored. For a number of sparse matrix kernels, it was shown that through effective exploitation of this search space an optimized code can be found that in most cases outperforms implementations of this routine found in sparse algebra libraries, but is at least on par in performance.

Chapter 11 introduced the *ready clause*, which allows dependencies between tuples to be described in a natural way. As a consequence, sets of tuples that can be processed in parallel can be deduced in a straightforward manner. Two execution models were presented for the parallel execution of *forelem* loops. Using these techniques, a parallel code for triangular solve could be automatically deduced and was shown to be competitive in performance to hand-optimized parallel implementations of triangular solve.

Finally, Chapter 12 discussed how distributed execution of *forelem* loops can be controlled by a compiler. Next to the optimization of local data storage formats that have been discussed in Part II of this thesis, this would give the *forelem* framework the capability to optimize the data decomposition and distribution as well. These capabilities were used to give an initial impression of the viability of the *forelem* framework to optimize Big Data applications.

Although two different classes of applications were discussed in this thesis, database applications and sparse matrix algebra, the optimization methodologies were based on a single intermediate representation. Many optimizations that were proposed for database applications in Part I of this thesis can also be applied to the irregular applications discussed in Part II of this thesis and vice versa. This gives rise to a lot of avenues for further research, a number of which will be briefly discussed in this chapter.

Based on the automatic global integrated optimization process that performs vertical integration of database applications, there are several directions for future work that are interesting from both a scientific and engineering point of view.

At the core of the global optimization process is the ability to express (SQL) queries in terms of *forelem* loops. The majority of SQL queries, including nested queries and group-by queries, can be written as *forelem* loops using the techniques described in this thesis. For a production system, future work is needed is support these parts of the SQL standard that cannot yet be expressed in the *forelem* intermediate representation.

Also the new common intermediary level, where application and data access codes are combined, should be further exploited. We intend to look into the application of existing code transformations at this new level as well as to investigate new basic code transformations that enhance the performance of the application. Existing transformations to be investigated include traditional loop transformations and vectorization. Furthermore, using the established technique of Def-Use analysis, elimination of unused query results and redundant queries is obtained for free. New basic code transformations will be researched, that can optimize code patterns found in database applications that cannot be handled by the existing compiler techniques. These transformations will rely on combining knowledge from both the application program and its queries. This may lead to new sophisticated techniques for the automatic optimization and merging of what were originally separate queries.

Chapter 3 proposed the Loop Collapse and Reverse Loop Collapse transformations. These transformations affect the schemas of the tables in addition to the loop structure of the code. By applying these transformations, the schema of the tables used by the application can be optimized based on the operations that are performed on the tables by the application code. The new intermediary level provides a good test bed to study the effectiveness of these transformations. Techniques can be developed to automatically optimize schemas of database tables, based on the different queries that are performed on these tables and the further processing of the data by the application code. Furthermore, the relation of these techniques with materialization discussed in Chapter 9 should be investigated.

In the context of sparse matrix algebra, we want to examine the effects of exploiting specific sparse matrix characteristics in the transformation process. This may result in the automatic generation of hybrid data storage formats. We also intend to investigate the effects of combining the loop blocking transformation with the materialization and concretization transformations. Materialization of a loop nest that is blocked should result in a blocked data storage format. The investigation should focus on whether further transformations can be devised to result in new forms of blocked data storage formats and the performance characteristics of the different formats should be explored.

The experimental evaluations presented in Part II of this thesis have focused on sparse matrix algebra. The optimization techniques that have been described are generic in nature, however. To demonstrate the effectiveness of our approach, we intend to conduct an extensive experimental evaluated of the proposed optimization techniques on a large variety of irregular codes. This will include further, more complicated, sparse matrix algebra routines, as well as routines from different domains such as graph algorithms.

Chapter 12 presented an initial overview of the extensions of the *forelem* framework to be able to control distributed execution of *forelem* loops. Through this control, different codes making use of different data decompositions and distributions can be generated automatically. The end goal is that within the *forelem* framework, from a single initial representation of a computation in the *forelem* intermediate representation, different variants can be generated for serial, locally parallel (multicore CPU or GPU), distributed and combined locally parallel and distributed execution. These different variants make use of different local data

storage formats and different data decompositions and distributions.

Methods for the automatic optimization of data decomposition and distribution within the *forelem* framework have not been investigated and remain a topic for future work. For Big Data applications it must be taken into account however that often the data to be processed is already stored and the data generation code is not part of the optimization process. The volume of the stored data may prohibit preprocessing, reformatting or redistribution of the data to better suit the computation. Strategies have to be investigated to find a middle ground between the reformatting of the existing data and the optimization of the computation. Note that, although many of such capabilities can be implemented in existing systems such as Hadoop, or are already implemented (e.g. binary storage in between jobs), there is at this moment no possibility for automatic optimization because many of these details are obscured from optimizing compilers.

At all levels, substantial improvements are possible in the code generators. For example: the code generator for CPU code can be extended with support for multi-core processing and SIMD instructions; the code generator for GPU code can be improved with optimizations to address coalescing, memory banking and interleaved computation; and the code generator for distributed codes can make better use of MPI by using better performing MPI primitives and to support interleaved computation and communication. Finally, the generated MPI code should also be made capable of fault tolerance by supporting continued execution if one of the nodes failed. Next to being able to handle a static loop schedule determined at compile-time, the code should be able to handle some amount of dynamic scheduling as well to allow for load balancing and failure recovery.

The *forelem* framework is presented as a versatile framework that unifies optimization of imperative and declarative codes. For the different application areas of this framework, different transformations have been devised. While such transformations were defined within the context of a particular application area, these transformations are generic in nature and are well of use in other application domains. For example, transformations that have been described in Part I in the context of database applications can also be applied on generic codes described in Part II.

Finally, it is interesting to see how techniques developed specifically for a single application domain can be used for the optimization of problems from another domain. So far, the focus of this research has been on the domains of database applications and sparse matrix algebra. It will be exciting to see whether investigations into other application domains will unveil further code transformations that can be expressed in a generic nature and that will subsequently result in improved performance of code from other application domains. This in particular shows the strength of the *forelem* framework as a versatile framework for the specification of code transformations and the optimization of tuple-based codes.

Bibliography

[1] Y. Ahmad and C. Koch. DBToaster: a SQL compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, 2009.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, Mar. 1976.

[3] J. R. Allen. *Dependence Analysis for Subscripted Variables and its Applications to Program Transformations*. PhD Dissertation, Rice University, 1983.

[4] J. R. Allen and K. Kennedy. Automatic loop interchange. In *ACM SIGPLAN Notices*, volume 19, pages 233–246. ACM, 1984.

[5] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9:491–542, October 1987.

[6] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 739–753, New York, NY, USA, 2010. ACM.

[7] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. *Data Engineering, International Conference on*, 0:821, 2003.

[8] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(01):73–95, 1989.

[9] H. Andrade, S. Aryangat, T. M. Kurç, J. H. Saltz, and A. Sussman. Efficient execution of multi-query data analysis batches using compiler optimization strategies. In *LCPC*, pages 509–524, 2003.

[10] C. Ashcraft and R. Grimes. Spooles: An object-oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, pages 22–27, 1999.

[11] A. Badia and D. Anand. Fighting redundancy in sql: the for-loop approach, 2004.

[12] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. Van Der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*, volume 11. Society for Industrial Mathematics, 1987.

[13] U. Banerjee. Unimodular transformations of double loops. *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, 1990.

[14] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[15] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*, pages 416–424. ACM, 1993.

[16] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB JournalThe International Journal on Very Large Data Bases*, 8(2):101–119, 1999.

[17] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.

[18] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 225–237, Asilomar, CA, USA, January 2005.

[19] J. Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *Euro-Par '98 Parallel Processing*, pages 377–382. Springer, 1998.

[20] S. Chaudhuri, V. Narasayya, and M. Syamala. Bridging the application and DBMS divide using static analysis and dynamic profiling. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 1039–1042, New York, NY, USA, 2009. ACM.

[21] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution. *Data Engineering, International Conference on*, 0:1284–1287, 2011.

[22] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[23] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[24] Computer Economics, Inc. Best Practices and Benchmarks in the Data Center, April 2006. Available at `http://www.computereconomics.com/article.cfm?id=1116`, accessed at July 2012.

[25] W. R. Cook and A. H. Ibrahim. Integrating programming languages and databases: What is the problem. In *In ODBMS.ORG, Expert Article*, 2005.

[26] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault. MPADS: memory-pooling-assisted data splitting. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 101–110, New York, NY, USA, 2008. ACM.

[27] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.

[28] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[29] U. Dayal, N. Goodman, and R. H. Katz. An extended relational algebra with control over duplicate elimination. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 117–123, New York, NY, USA, 1982. ACM. Available at `http://doi.acm.org/10.1145/588111.588132`.

[30] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[31] S. W. Dietrich and M. Chaudhari. The missing LINQ between databases and object-oriented programming: LINQ as an object query language for a database course. *J. Comput. Small Coll.*, 24(4):282–288, 2009.

[32] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. 1994.

[33] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. *SIGARCH Comput. Archit. News*, 35:13–23, June 2007.

[34] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In B. Pugh and C.-W. Tseng, editors, *Languages and Compilers for Parallel Computing*, volume 2481 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 2005.

[35] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *J. Parallel Distrib. Comput.*, 5(5):587–616, 1988.

[36] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 23–33, New York, NY, USA, 1987. ACM.

[37] C. Garrod, A. Manjhi, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic Application Analysis for Update-Independence. Paper 1107, 2008. Available at `http://repository.cmu.edu/compsci/1107`.

[38] A. Gawecki and F. Matthes. *Integrating query and program optimization using persistent CPS representations.*, pages 496–501. ESPRIT Basic Research Series. Springer Verlag, 2000.

[39] Gil, Joseph (Yossi) and Lenz, Keren. Eliminating impedance mismatch in C++. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1386–1389. VLDB Endowment, 2007.

[40] P. Gottschling, D. S. Wise, and M. D. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125. ACM, 2007.

[41] D. Habich, S. Richly, and W. Lehner. GignoMDA: exploiting cross-layer optimization for complex database applications. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1251–1254. VLDB Endowment, 2006.

[42] HipHop for PHP Project. HipHop for PHP, 2012. Available at `https://github.com/facebook/hiphop-php/wiki/`, accessed at July 2012.

[43] K. Iglberger, G. Hager, J. Treibig, and U. Rude. High performance smart expression template math libraries. In *High Performance Computing and Simulation (HPCS)*, pages 367–373. IEEE, 2012.

[44] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. *Computational Science ICCS 2001*, pages 127–136, 2001.

[45] ISO/IEC 9075:1992. Information technology – Database languages – SQL, 1992.

[46] Jonathan G. Koomey. Growth In Data Center Electricity Use 2005 To 2010. Aug 2011.

[47] M. H. Kang, H. G. Dietz, and B. K. Bhargava. Multiple-query optimization at algorithm-level. *Data Knowl. Eng.*, 14(1):57–75, 1994.

[48] Kees Verstoep. DAS-4: Distributed ASCI Supercomputer 4, 2013. Available at `http://www.cs.vu.nl/das4/`, accessed at January 2013.

[49] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[50] K. Kennedy. *A survey of data flow analysis techniques*, pages 5–54. Prentice-Hall, Englewood Cliffs NJ, 1981.

[51] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 20(4):869–916, July 1998.

[52] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer Berlin / Heidelberg, 1994.

[53] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, Sept. 1982.

[54] P. Knijnenburg, T. Kisuki, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.

[55] V. Kotlyar, K. Pingali, and P. Stodghill. A Relational Approach to the Compilation of Sparse Matrix Programs. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par*, volume 1300 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 1997.

[56] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th conference on Computing frontiers*, pages 87–96. ACM, 2008.

[57] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[58] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[59] K. S. Kundert et al. Sparse matrix techniques. *Circuit Analysis, Simulation and Design*, 3(pt 1), 1986.

[60] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS*, pages 63–74, 1991.

[61] D. F. Lieuwen. Parallelizing Loops in Database Programming Languages. In *ICDE*, pages 86–93, 1998.

[62] D. F. Lieuwen and D. J. DeWitt. A Transformation-Based Approach to Optimizing Loops in Database Programming Languages. In *SIGMOD Conference*, pages 91–100, 1992.

[63] LLVM Project. The LLVM Compiler Infrastructure Project. Available at `http://www.llvm.org/`, accessed at February 2013.

[64] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 600–611, New York, NY, USA, 2002. ACM.

[65] D. Maier. *Representing database programs as objects*, pages 377–386. ACM, New York, NY, USA, 1990.

[66] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic Query Transformations for Dynamic Web Applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1175–1178, Washington, DC, USA, 2009. IEEE Computer Society.

[67] B. Marker, J. Poulson, D. Batory, and R. Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 362–378. Springer Berlin Heidelberg, 2013.

[68] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 14th international conference on Supercomputing*, pages 88–99. ACM, 2000.

[69] MonetDB. MonetDB. Available at `http://www.monetdb.org/`, accessed at February 2013.

[70] MySQL Project. libmysqld, the Embedded MySQL Server Library, 2012. Available at `http://dev.mysql.com/doc/refman/5.6/en/libmysqld.html`, accessed at July 2012.

[71] P. Nagpurkar, W. Horn, U. Gopalakrishnan, N. Dubey, J. Jann, and P. Pattnaik. Workload characterization of selected JEE-based Web 2.0 applications. In *IISWC*, pages 109–118, 2008.

[72] M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. Technical report, NVIDIA Technical Report, NVR-2011-001, 2011.

[73] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4:539–550, June 2011.

[74] ObjectWeb Consortium. JMOB - RUBBoS Benchmark, 2005. Available at `http://jmob.ow2.org/rubbos.html`, accessed at July 2012.

[75] ObjectWeb Consortium. RUBiS - Home Page, 2008. Available at `http://rubis.ow2.org/`, accessed at July 2012.

[76] Oracle. Oracle In-Memory Database Cache, 2011. Available at `http://www.oracle.com/us/products/database/in-memory-database-cache-066510.html`, accessed at July 2011.

[77] Padua, David A. and Wolfe, Michael J. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, Dec. 1986.

[78] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with satellite databases. *The VLDB Journal*, 17(4):657–682, July 2008.

[79] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Computers, IEEE Transactions on*, 100(12):1425–1439, 1987.

[80] PostgreSQL Project. PostgreSQL: The world's most advanced open source database. Available at `http://www.postgresql.org/`, accessed at February 2013.

[81] R. Pozo and K. Remington. Sparselib++ v. 1. 5 sparse matrix class library. reference guide. *NASA*, (19980018804), 1996.

[82] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[83] K. F. D. Rietveld and H. A. G. Wijshoff. Forelem: A Versatile Optimization Framework For Tuple-Based Computations. In *CPC 2013: 17th Workshop on Compilers for Parallel Computing*, July 2013.

[84] K. F. D. Rietveld and H. A. G. Wijshoff. Quantifying Energy Usage in Data Centers Through Instruction-Count Overhead. In *SMARTGREENS 2013, 2nd International Conference on Smart Grids and Green IT Systems*, pages 189–198, May 2013.

[85] K. F. D. Rietveld and H. A. G. Wijshoff. To Cache or Not To Cache: A Trade-off Analysis For Locally Cached Database Systems. In *ACM International Conference on Computing Frontiers*, May 2013.

[86] K. F. D. Rietveld and H. A. G. Wijshoff. Towards A New Tuple-Based Programming Paradigm for Expressing and Optimizing Irregular Parallel Computations. In *ACM International Conference on Computing Frontiers*, May 2014. Accepted for publication.

[87] Y. Saad. *SPARSKIT: A basic tool kit for sparse matrix computation*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[88] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of Caching and Replication Strategies for Web Applications. *IEEE Internet Computing*, 11:60–66, 2007.

[89] J. G. L. Steele. RABBIT: A compiler for SCHEME. Technical Report AITR-474, MIT Artificial Intelligence Laboratory, May 6 1978.

[90] C. Stewart, M. Leventi, and K. Shen. Empirical examination of a collaborative web application. In *IISWC*, pages 90–96, 2008.

[91] Transaction Processing Performance Council. TPC-H, May 2009. Available at `http://tpc.org/tpch/default.asp`, accessed at November 2011.

[92] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *Parallel and Distributed Systems, IEEE Transactions on*, 4(1):87–98, 1993.

[93] U.S. Environmental Protection Agency. Report to Congress on Server and Data Center Energy Efficiency. Aug 2007.

[94] H. L. A. van der Spek, S. Groot, E. M. Bakker, and H. A. G. Wijshoff. A compile/run-time environment for the automatic transformation of linked list data structures. *Int. J. Parallel Program.*, 36(6):592–623, Dec. 2008.

[95] H. L. A. van der Spek, C. W. M. Holm, and H. A. G. Wijshoff. How to unleash array optimizations on code using recursive data structures. In *ICS*, pages 275–284, 2010.

[96] G. Wassermann, C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[97] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 307–316, New York, NY, USA, 2006. ACM.

[98] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[99] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

[100] M. Wolfe. Vector optimization vs vectorization. *Journal of Parallel and Distributed Computing*, 5(5):551 – 567, 1988.

[101] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 419–430, New York, NY, USA, 2009. ACM.

[102] D. P. Yach, J. D. Graham, and A. F. Scian. Database system with methodology for accessing a database from portable devices. US Patent #6341288, Jan 2002.

[103] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *ACM SIGPLAN Notices*, volume 39, pages 255–266. ACM, 2004.

[104] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. ACM, New York, NY, USA, 1991.

Samenvatting

Een veelzijdig optimalisatieraamwerk gebaseerd op tupels

Het hart van een computer is de Centrale Verwerkingseenheid (CVE, Engels: CPU, Central Processing Unit). Deze eenheid voert programma's uit welke zijn opgebouwd uit elementaire instructies zoals het lezen van data uit het geheugen en het uitvoeren van simpele operaties op data. Het opstellen van programma's in dit soort elementaire instructies is een zeer tijdrovend proces. Dit heeft geleid tot de ontwikkeling van programmeertalen, waarin programma's op een hoger, abstracter niveau kunnen worden uitgedrukt. Met behulp van software kunnen programma's die zijn geschreven in zo'n programmeertaal automatisch worden vertaald naar een machinecode die door de Centrale Verwerkingseenheid kan worden uitgevoerd.

Programmeertalen kunnen worden ingedeeld in verschillende paradigma's. Twee paradigma's die in dit proefschrift centraal staan zijn imperatieve programmeertalen en declaratieve programmeertalen. Met imperatieve programmeertalen wordt de berekening die moet worden uitgevoerd stap voor stap beschreven. Er wordt dus exact vastgelegd hoe de berekening moet worden uitgevoerd. Daarentegen wordt met declaratieve programmeertalen vastgelegd *wat* moet worden gedaan, bijvoorbeeld welke data moet worden opgehaald uit een database, maar niet hoe en niet in welke volgorde.

Tijdens de vertaalslag naar machinecode kunnen programma's worden aangepast zodat deze efficiënter werken, onder de voorwaarde dat de semantiek/betekenis van het programma ongewijzigd blijft. Deze efficiëntere machinecodes voeren het programma uit in minder tijd of werkgeheugen. In de praktijk is de manier waarop deze optimalisaties worden uitgevoerd afhankelijk van het programmeerparadigma. Bij de optimalisatie van declaratieve programma's ligt de nadruk op het bepalen van een efficiënt stappenplan voor het ophalen van de gespecificeerde data. Dit stappenplan bestaat uit een aaneenschakeling van aanroepen naar vooraf vastgelegde routines die al in machinecode zijn uitgedrukt. Imperatieve programma's worden in de regel vertaald naar machinecode met een "compiler". Geavanceerde compilers zijn uitgerust met transformaties. Dit zijn transformaties als het herordenen van de stappen die door het programma worden uitgevoerd of het selecteren van efficiënte instructies om de benodigde berekeningen uit te voeren. Deze transformaties hebben als doel specifieke functionaliteiten en karakteristieken van het beoogde hardwareplatform beter te benutten. Compi-

lers die in staat zijn dit soort optimaliserende transformaties toe te passen worden "Optimizing Compilers" genoemd.

In de regel worden applicatieprogramma's geschreven in een imperatieve programmeertaal. Specificaties voor het ophalen van data uit een database worden veelvuldig geschreven in een declaratieve programmeertaal. Een voorbeeld van zo'n declaratieve taal is SQL en specificaties geschreven in SQL worden "queries" genoemd. Database applicaties zijn applicatieprogramma's die data verwerken welke zijn opgeslagen in een database. Zulke applicaties zijn dus opgebouwd uit zowel imperatieve als declaratieve codes. Deze codes worden onafhankelijk van elkaar geoptimaliseerd op verschillende manieren en in verschillende modules. Dus, de declaratieve codes die data ophalen worden onafhankelijk geoptimaliseerd van imperatieve codes die de opgehaalde data verder verwerken.

Het eerste deel van dit proefschrift richt zich op database applicaties en in het bijzonder op web applicaties. Web applicaties verzorgen interactieve websites, waarop bijvoorbeeld reizen kunnen worden geboekt of aankopen gedaan. Deze applicaties zijn in het algemeen modulair opgebouwd: naast de applicatie zelf wordt er gebruik gemaakt van een web server en een databasesysteem (DBMS: Database Management System). Door deze modulariteit kunnen web applicaties in een (zeer) kort tijdbestek worden geïmplementeerd. Echter, deze modulariteit heeft zijn prijs. Door de verschillende modules met elkaar te integreren kunnen deze applicaties vele malen efficiënter worden gemaakt. 90% van de machinecode instructies kunnen worden geëlimineerd zonder dat dit het eindresultaat beïnvloed. Naast een significante verhoging in snelheid leidt dit ook tot een besparing van energie: minder servers zijn nodig om eenzelfde aantal bezoekers te kunnen verwerken.

Het integreren van deze verschillende modules betekent ook dat de verschillende methodologiën om declaratieve en imperatieve programma's te optimaliseren moeten worden samengevoegd. In dit proefschrift wordt een raamwerk geïntroduceerd om deze integratie te faciliteren: het *forelem* raamwerk. Het *forelem* raamwerk biedt een generieke methode om declaratieve codes voor de toegang tot data uit te drukken in codes gebaseerd op imperatieve constructies zoals simpele loops (de *forelem* loop) en benaderingen van arrays. Dit maakt het mogelijk oorspronkelijk declaratieve codes te mengen met imperatieve codes. Een belangrijke consequentie hiervan is dat traditionele analyses voor imperatieve codes, zoals het ontdekken van ongebruikte resultaten, alsook traditionele imperatieve transformaties, kunnen worden toegepast op deze gecombineerd codes. Hiermee wordt het aantal optimalisatiemogelijkheden sterk vergroot.

Hoewel het *forelem* raamwerk in eerste instantie is ontwikkeld voor database applicaties is het door de generieke aard ook toepasbaar in andere gebieden. Omdat het raamwerk is ontworpen voor database applicaties, is het opgebouwd rond het wiskundige concept van tupels. Tupels zijn echter zeer geschikt als elementaire data representatie voor imperatieve codes, omdat tupels de meest fundamentele objecten zijn om met elkaar gerelateerd waarden te koppelen.

In het tweede deel van dit proefschrift wordt het gebruik van het *forelem* raamwerk bestudeerd voor het optimaliseren van irreguliere applicaties. Irreguliere applicaties zijn moeilijk te optimaliseren door compilers, omdat het verloop van de berekening van te voren niet geheel vast staat. Een optimalisatiemethode

wordt beschreven waarbij data gebruikt door een berekening wordt vertaald naar tupels en de berekening zelf uitgedrukt wordt als *forelem* loops die deze tupels verwerken en manipuleren. Op deze manier kan zowel de berekening worden herordend alsmede de manier waarop tupels worden opgeslagen en gestructureerd. In feite leidt dit tot de automatische generatie van datastructuren door een compiler. Deze technieken worden bestudeerd aan de hand van sparse matrix berekeningen. Sparse matrices kennen vele toepassingen in het wetenschappelijk rekenen en technische wetenschappen, zoals circuit simulaties en vloeistofdynamica.

Om het raamwerk toepasbaar te maken voor meerdere klassen van irreguliere applicaties, wordt een uitbreiding beschreven voor het coderen van data afhankelijkheden. Een conditie kan worden aangegeven dat een tupel pas mag worden verwerkt als tupels die aan de conditie voldoen al verwerkt zijn. Ten slotte beschrijft dit proefschrift een uitbreiding voor het uitdrukken van gedistribueerde berekening in het *forelem* raamwerk.

Het *forelem* raamwerk dat is beschreven in dit proefschrift is in staat om zowel traditionele imperatieve codes (zoals sparse matrix berekeningen) en declaratieve codes (zoals database queries) te optimaliseren. Dit maakt het raamwerk veelzijdig en in vele gebieden toepasbaar. Ook kunnen optimalisatietechnieken die zijn ontwikkeld in een bepaald gebied worden toepast in andere gebieden, iets dat zonder onderliggend generiek raamwerk niet mogelijk is.

Acknowledgments

I want to thank Mattias Holm and Harmen van der Spek for the interesting discussions during my PhD research. In addition, I would like to thank other colleagues at LIACS with whom I worked together on teaching courses and in the computer committee: Teddy, Fons, Hendrik-Jan and Frank.

My lovely partner Ginny deserves a lot of thanks for her love and profound understanding when I had to work through the evening and sometimes night. She has taught me to believe in myself and unconditionally supported me during my research. Thank you so much!

I am grateful to my parents, Krijn and Diana, for creating an environment wherein I could safely grow up and learn to discover. They encouraged me to build, take apart and question: techniques that have proved to be very useful during my research. Your unequivocal support means a lot to me.

Thanks to Laura van As for her advice on improving the cover design.

Several friends deserve to be acknowledged for their interest in my work, their support and their interesting on- and off-topic discussions: my two brothers Alexander and Martijn, Thijs, Wouter, Pieter, Vian, Robert, Ron, Christian, Mitch, Carlos and many others.

Curriculum Vitae

Kristian Rietveld was born in Vlaardingen, the Netherlands, in 1984. He quickly got interested in building things, initially with Lego bricks, but by the end of primary school also virtually by programming computers. In July 2002, he obtained a Gymnasium diploma at the Stedelijk Gymnasium in Schiedam. During his last two years at the Gymnasium, he became part of the development team of the GTK+ open source project, which underpins popular Linux applications such as the GIMP and the GNOME desktop. From the onset it was clear Kristian would go on to study Computer Science. He did so at Leiden University, receiving a Bachelor's degree in Computer Science in March 2009 and a Master's degree in August 2009. During his studies at Leiden University, he worked as part of the system administrator team at the Mathematical Institute of Leiden University for two years. He also continued to be involved in the GTK+ project. This lead Kristian to be hired by the former open source consulting company Imendio in 2005. Within Imendio, he worked as a contractor for Nokia on the Maemo platform, a software platform for Internet tablets and smartphones. After a Master's research project, his interest in Computer Systems research was sparked. As a result, he started as a PhD student in the High Performance Computing group at the Leiden Institute for Advanced Computer Science (LIACS) in October 2009. During his research, he worked on a framework for the vertical integration of database applications and evaluated other application areas in which this framework can be used. He also wrote a small operating system kernel for the ARM architecture to be used in the Operating Systems Bachelor course, together with Mattias Holm. Next to his research, Kristian also continued to work as a Lead Software Engineer for Lanedo, that took over Imendio's business in the beginning of 2009. During his time at Lanedo Kristian worked on projects for Samsung R&D and Xamarin. Since November 2013, Kristian is a Researcher and Lecturer at LIACS.