# Transparent restructuring of pointer-linked data structures

Spek, H.L.A. van der

# Transparent Restructuring of Pointer-Linked Data Structures

Harmen Laurens Anne van der Spek

# Transparent Restructuring of Pointer-Linked Data Structures

Proefschrift

**ter verkrijging van**
**de graad van Doctor aan de Universiteit Leiden,**
**op gezag van Rector Magnificus prof. mr. P.F. van der Heijden,**
**volgens besluit van het College voor Promoties**
**te verdedigen op dinsdag 7 december 2010**
**klokke 13.45**

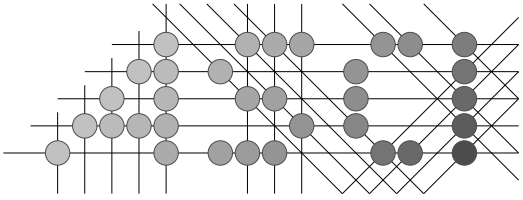
door

Harmen Laurens Anne van der Spek

geboren te Zevenhuizen in 1982

## Promotiecommissie:

| | |
|---|---|
| Promotor: | Prof. dr. H.A.G. Wijshoff |
| Copromotor: | Dr. E.M. Bakker |
| Overige leden: | Prof. dr. W. Jalby (Université de Versailles) |
| | Prof. dr. B.H.H. Juurlink (Technische Universität Berlin) |
| | Prof. dr. J.N. Kok |
| | Prof. dr. F.J. Peters |



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 220.

Transparent Restructuring of Pointer-Linked Data Structures
Harmen Laurens Anne van der Spek
PhD Thesis, Universiteit Leiden
ISBN: 978-90-8891-216-0
Printed by: Proefschriftmaken.nl
Published by: Uitgeverij BOXPress, Oisterwijk

*To my wife Erika*

# Contents

# CHAPTER 1

# Introduction to the Introduction

The development of digital computers started in the previous century. At first, such systems were programmed by hand, at a very low level. The need for abstraction was soon recognized, and programming languages and tools to simplify the programming process were developed with success. The concepts developed at that time are still heavily used, although the complexity of both hardware and software has increased dramatically. In this chapter, an overview of the current state of the art technologies concerned with high performance computing is presented. We will address contemporary processors and the memory hierarchy with its inherent problems. The software-based technologies to make efficient use of these computers can be divided into three groups: compilers, programming languages and (run-time) libraries. Each of these topics is discussed and related to the latest developments in processor technology: the multi-processor on a chip.

## 1.1 Contemporary Processors

Parallel computer systems are by no means a new phenomenon. Until recently, however, these systems were not widely deployed and used. This changed in the last decade, when merely pushing single-threaded performance to its limits basically became a dead end. Gordon Moore's famous law, that the number of transistors on an integrated circuit doubles every two years (his first estimate was every year), still holds. This abundance of transistors is put to use in multi-core processors, combined with large caches. Other developments include the heterogeneous architecture, such as the IBM Cell and the throughput-oriented platform such as NVIDIA's and AMD's GPUs.

Simply put, transistors are used to replicate components. These new platforms

provide immense computational power, but each of them requires the programmer to write code that specifically targets these platforms. Essentially, the introduction of these platforms did not raise a new question on how to program parallel systems. This question was relevant before, but the widespread introduction of parallel systems have made this same question more relevant than ever before.

The Intel 4004, the first fully integrated microprocessor, was launched by Intel in 1971. It consisted of 2300 transistors. Over the last few decades, the transistor densities have increased dramatically, nicely following Moore's law. The Intel Core i7-960 for example consists of 731 million transistors. The latest NVIDIA GPU Fermi architecture sports 3 billion transistors [95].

For decades, the availability of more and more transistors led to the integration of many components onto the chip. Processors became pipelined and caches were introduced to mitigate the effects of the ever growing gap between processor speed, and memory bandwidth and latency. Pipelining in turn led to the introduction of branch predictors, to prevent pipeline stalls. Patt discusses these and more driving factors behind the progression in the field of microprocessors [98].

In this section, we will briefly consider some of the most important techniques and concepts that are found in today's high-end processors. All these different components make the design and optimization of high-performance software a complicated task compared to the time in which a single instruction was fetched, executed and retired in one cycle. As the most prevalent architecture today is the Intel x86 architecture, we will use mainly this architecture to illustrate the developments in processor technology in the last two decades.

### 1.1.1 Pipelining

Pipelining is a technique that segments a large operation into multiple sub-operations. Naturally, each sub-operation is smaller than the entire large operation, and thus the cycle time can be reduced. For example, the MIPS pipeline used as an illustration by Hennessy and Patterson [53] consists of five stages: instruction fetch, instruction decode, execute, memory operation and write back. As soon as the first instruction has been fetched, the next instruction can be fetched in the next cycle, while the first instruction proceeds with the decode stage. Once the pipeline is fully filled, a theoretical increase in performance of a factor 5 can be achieved versus non-pipelined execution. In practice, this is not the case. Due to data dependencies, branching behavior and lack of resources the pipeline might need to be flushed.

The Intel Pentium processor features two integer pipelines with a depth of five. It includes a prefetch stage, two decode stages, an execute stage and a write back stage. These two pipelines can execute two instructions in parallel, if the two instructions meet certain requirements [44]. The Pentium Pro, II and III take a different strategy. These processors issue the instructions in-order, decode them into smaller micro-ops, which are executed in an out-of-order core. The full pipeline consists of the following stages: branch predictions (2 stages), instruction fetch (3 stages), instruction decode (2 stages), register alias table, micro-op reorder buffer for reads, reservation stations (in which the micro-ops wait until their operands are available), multiple execution

ports to which various micro-ops can be issued, write back to the reorder buffer and the register retirement file (for in order retirement) [44, 61]. The Pentium M is a design that is based on the architecture of the Pentium Pro/II/III described above.

The newer Core 2 and Core i7 designs are also a member of this family. The changes mostly aim at higher throughput by increasing the number of micro-ops that can be executed per clock cycle. Of course, such increased throughput capabilities require that other stages can keep up, so for example, the instruction decoding must be able to keep up and provide enough micro-ops to execute.

An interesting member of the Intel x86 family is the Pentium 4, which has a different design from the other processors described above. The Pentium 4 featured very deep pipelines, which allowed it to use very high clock rates. This led to excessive power consumption, which is one of the reasons why this architecture has been abandoned by Intel. Its successors, the Core 2 and Core i7 are based, as stated earlier, on the Pentium M line. One of the interesting things about the Pentium 4 was the use of Hyper Threading, which allowed the pipeline to be kept full by issuing instructions from two different instruction streams into a single execution pipeline. This feature was not present in the later Core 2 architecture, but has been reintroduced in the latest Core architectures (i3, i5 and i7).

Around the same time of the Pentium 4, AMD approached the problem from a different angle. Instead of building their processors using very deep pipelines, they used three parallel pipelines which are able to handle almost any operation. The AMD processors do not decode the instruction in micro-ops, but rather use more high-level macro-ops, that are decomposed as late as possible.

As can be seen, there is a great diversity in the implementation of pipelines. For example, Hennessy and Patterson's MIPS pipeline [53] is very different in structure from the Intel and AMD pipelines. Avoiding stalls in such complicated pipelines is not an easy task, and compilers preferably should take the actual pipeline implementation into account when generating code, but this is unfeasible in many practical cases. For example, a software vendor will not provide a different installation package for each different processor on the market, especially not for consumer software.

### 1.1.2 Multi-Core Processors

The abundance of transistors must be put to use somehow. As shown above, much of this is used to speedup single-threaded execution using pipelining and other structures that improve performance such as branch prediction. Most of the real estate, though, is used for caches (Section 1.1.3). Eventually, adding more complexity to the architectures turned out not to be the best way forward. Instead of pursuing more instruction level parallelism, the engineers moved their focus to putting multiple cores on a single chip. The IBM POWER4 was the first chip to include two cores on a single chip. Intel started to produce dual-core processors in their Core line, the Core Duo being the first with two cores on a chip. The Core 2 line also included 2 cores per chip. The quad core version does not have 4 cores on a single chip, but it is a composite of two dual core chips. Real quad cores and 6 core machines are found in the Intel

Core i7 line. Each of these cores can execute 2 threads, resulting in a maximum of 12 concurrently running threads.

Their tight integration allows for low latency communication between the different cores. This is a major difference between on chip multi-core systems and the traditional multi-processor systems, where each processor was put in its own socket. For executing some independent processes, this is fine, as long as the joint memory bandwidth is not exceeded. The main problem is that the processors themselves are very fast and as long as each processor is not interfering with another processor, things are fine. In practice, processes do need to fetch data from main memory, and eventually some resources must be shared, such as L2 or L3 caches and the memory bus. So, while there is great potential in the processing capabilities of multi-core systems, it is an art to write applications that make full (or even reasonable) use of these vast computing resources.

### 1.1.3 The Memory Hierarchy

In early processors, the speed of the processor and the attached main memory storage were roughly similar, and as a consequence, accessing main memory did not result in large penalties. However, with the advances in computer architecture and transistors getting smaller and smaller, the increase in performance of CPUs has outperformed the decrease in memory latency by several orders of magnitude (Hennessy and Patterson show a difference of over a factor of 1000 in 2010 [53] with the year 1980 as reference point).

To overcome the performance gap between processors and main memory, caches were introduced, which are small, but fast, memories that are close to the processor. Today, all high-performance general purpose CPUs have at least most of their cache levels integrated on the chip. With the ever growing gap between the performance of processors, and memory bandwidth and latency, single levels of cache have been extended to multiple cache levels, and in many multi-core processor designs, the caches closer to main memory are a shared resource. For example, the cores of the Intel Core 2 Duo and later processors share parts of the caches, and can even dynamically change the fraction allocated to a particular core. Such autonomous behavior (this is not programmer controlled) can affect the performance of programs in an unpredictable way, and hence it is very challenging to optimize for such architectures. This also led to research in the field of scheduling, were active co-scheduling of jobs is used to increase throughput. Note that next to performance issues, there is also the question of security and reliability. Moscibroda and Mutlu have shown that the performance of programs can be negatively affected by other processes that are especially crafted to interfere with co-scheduled processes [88].

Similar to the complicated pipelines and the advances in multi-core processors, caches pose a significant challenge to programmers of high-performance applications. On the one hand, the transparency of the memory subsystem is a good abstraction which frees to programmer from the responsibility where to store data. On the other hand, the lack of control also implies that one must accept the unpredictable nature of caches, especially if co-scheduling of other jobs is taken into account. Software

controlled caches (also known as scratchpads) have been implemented to give the programmer explicit control over what should be in the cache and what not. The Cell processor [56], developed by Sony, Toshiba and IBM, consists of one PowerPC core, connected to several (6 on the PlayStation 3, 8 on the blade systems) so called Synergistic Processing Elements (SPEs). Each of this SPEs has its own local scratchpad memory (256KiB) which needs to be explicitly controlled using DMA transfers. The recently announced NVIDIA Fermi architecture [95] contains 16 streaming multiprocessors, each of which contains 32 cores. Each streaming multiprocessor has its own local memory with a size of $64KiB$. This can either be decomposed into $48KiB$ of shared memory (NVIDIA's terminology for scratchpad memory) and $16KiB$ of L1 cache, or $16KiB$ of shared memory and $48KiB$ of L1 cache [95]. Where its predecessors did not have an L2 cache, the Fermi architecture features an L2 cache of $768KiB$. Over time, it can be said that the memory architecture of GPUs is growing towards that of the general CPUs.

For general-purpose multi-chip CPUs, the common approach consists of providing fully coherent caches. Intel's Larrabee project [108] aims to put many simple x86 cores on a single chip, that have coherent caches. By providing cache control instructions cache lines can be marked for early eviction. They claim this allows a programmer to use the L2 cache similar to a scratchpad. It is unclear whether this coherent cache design will scale to larger many-core systems. All approaches such as directory-based coherence, snooping and snarfing suffer from the fact that transportation of data takes time, and consumes relatively large amounts of power. Also, the hardware costs increase quadractically with respect to the number of cores.

## 1.2 Software for Parallel Systems

Programming languages, (run-time) libraries and compilers form the set of tools available to implement systems. Programming languages and libraries serve as layers of abstraction to ease software development. The compiler is used to provide the translation from the higher level language to the lower level instruction set architecture (ISA). In the beginning, the focus was on automatic translation from higher level languages into machine specific code. Later, this focus moved to optimizing the resulting output code. While automatic parallelization has been a subject of research, the advent of mass-produced multi-core systems has made the subject of parallelization more important than ever.

Eigenmann and Hoeflinger state that there are three ways to create a parallel program [42]:

1. Writing a serial program and compiling it with a parallelizing compiler.
2. Composing a program from modules that have already been implemented as parallel programs.
3. Writing a program that expresses parallel activities explicitly.

In this section, we describe compilers, which mostly focus on Option 1, and programming languages and (run-time) libraries, which mostly fit the description of Option 2 and 3.

### 1.2.1 Compilers

The basic task of a compiler is to translate an input program written in a particular language into an output program. The output program can be expressed in another language or in the same language. In the latter case a compiler is often referred to as a source-to-source compiler. The first compilers (among which the first Fortran compiler implementation by Backus et al. [15]) greatly aided in easing development efforts of programs. Instead of retargeting an application to a new platform, the compiler would need to be extended to support the new platform, after which all existing codes can be recompiled for the new target platform. Modern compilers usually use a strategy in which different, source language dependent front-ends compile source programs into a common intermediate language. This common intermediate language can in turn be compiled to a binary program for a specific architecture.

Today, compilers do much more than the basic translation of a source program into a target program. Code optimization has become one of the major components of modern compilers. Examples of such optimizations are: inlining, loop optimizations, common sub expression elimination and inserting prefetching instructions. Especially loop optimizations are essential in obtaining high performance on many computationally intensive applications. Zima and Chapman provide an overview of such well-known techniques [127]. Optimizations that have been applied usually follow the developments in computer architecture. With the introduction of vector processors, vectorizing transformations were needed to exploit these new features. In the new, multi-core era parallelization is the key word, and new ways to compile for these architectures must be sought.

Simply put, there are two different factors in compiler design and implementation. The first factor is the driving force of advances in the field of compilation: features of the target architecture. For example, the introduction of vector processors needed compiler support, as otherwise, all existing code would have to be rewritten by hand to use these new vector instructions. Another example is automatic parallelizing transformations, which have been developed to exploit parallel architectures. The other major factor in compiler design and implementation is *code analysis*. Without proper analysis techniques, correctness and safety of transformations cannot be proved. Dependence analysis techniques are among the most important analyses found in parallelizing compilers [16, 17, 30, 78, 83, 97, 101]. Modern compilers include advanced reordering transformations based on dependence tests. CLooG is a code generator that use the polyhedral model [18] for dependence analysis. GCC includes its GRAPHITE framework [109], which uses CLooG/PPL.

As mentioned above, the memory hierarchy is a very important factor in performance, and therefore many locality-improving transformations have been proposed. Most of those transformations focus on repartitioning the iteration space in such a

way that the semantics of the program is preserved, but locality is increased. Examples of loop transformations are [127]: loop interchange [13], fission, fusion [66], unrolling, tiling [121] and skewing [122], to name a few. By improving locality, such optimizations can have a great effect on performance. The loop transformations focus on reordering computations. Obviously, we can also try to reorder data in memory to improve performance [64, 68, 96].

For large applications it is even more difficult to reorder computations and data layout. On the other hand, optimization can be much more effective, if the entire program is taken into account. GCC [1] and the Intel C++ compiler both support whole-program analysis. The LLVM compiler infrastructure [74] provides link time optimization, where code can be optimized after modules have been linked. Whole-program analysis combined with escape-analysis (which determines whether data might be used outside the current compilation unit) allows for determination of type-safety properties for type-unsafe languages [72]. The whole-program view enables far more aggressive optimization techniques that cannot be applied otherwise.

A special class of compilers is that of automatic parallelizing compilers. Traditionally, these have been designed and implemented for the Fortran language, such as the Polaris compiler [28] and the Vienna Fortran compiler [21]. Many Fortran codes show quite regular behavior with respect to their control flow structures. Especially in dense computations, in which arrays are directly accessed by access functions that only depend on the loop counters, the loops can in many cases be fully analyzed and parallelized at compile-time. The earlier mentioned polyhedral model has been very successful in determining dependencies in loops whose iteration space can be described by polyhedra. One major reason that automatic parallelization of Fortran code has been very successful, compared to other languages such as C and C++, is the fact that dependence analysis is easier in Fortran. This is a result from the common practice of defining the data regions used in the program at compile-time. This especially holds for code written in standard Fortran 77, which does not support dynamic memory allocation.

Fortunately for today's programmers, but unfortunately from the compiler perspective, dynamic memory allocation is widely used in languages such as C, C++, and Java. Dynamic memory allocation is also often used in conjunction with recursive data structures, such as tree and graph structures. The use of pointers that point to the heap (the memory area used to dynamically allocate data from) gives rise to the pointer aliasing problem. If two pointers to memory point to the same location, they are said to be aliased. In general, pointer analyses are not able to answer this question for every pair of pointers at compile-time, and may give three different answers to a query, if two pointers (or addressing expressions) are aliased: *pointers do not alias*, *pointers may alias* or *pointers must alias*. If a program is multi-threaded, the aliasing problem becomes even more complicated. In the next chapter, an overview of previous work on pointer analysis is provided.

For pointer-based codes, many of the techniques that have just been mentioned are either not sufficient, or cannot be applied. Typically, code using pointer-linked structures uses data dependent branch conditions in their loop header. Such loops cannot

be described in the polyhedral model. In addition, techniques like array-privatization cannot be applied, because languages like C do not guarantee anything about the location of allocated data for data elements in pointer-linked data structures. As a result, parallelization of such code to non-shared memory architectures is a nontrivial task that may require a substantial amount of handwork to translate structured data to different address spaces.

In order to guide the parallelization of code, OpenMP [8] provides compiler directives that are used to specify the parallel properties of code. These directives are then used by the compiler to produce a parallel implementation. While OpenMP can be used to express some parallel properties when using pointer-based codes (for example to traverse disjoint paths in a tree concurrently), in general, it can be stated that optimization and parallelization of applications using pointer-based structures have been relatively unsuccessful. The more recent CUDA [94] and OpenCL [67] frameworks can be regarded as a combination of compiler and library techniques to enable the definition of parallel algorithms.

### 1.2.2 Languages and (Run-Time) Libraries for Parallel Programming

It would be highly desirable if compilation of sequentially expressed code would result in automatically parallelized code that runs efficiently on any platform. Alas, this is not the case with today's compilers. Thus, one must resort to other solutions, that fit into the category of Option 2 and 3 stated by Eigenmann and Hoeflinger [42], which expresses that a program is built from components implemented as parallel programs, or that a program explicitly expresses parallelism. Two means can be distinguished that support these two options: programming languages and (run-time) libraries. We will briefly review a number of programming languages and libraries used to express and support the implementation of parallel applications.

#### Programming Languages

One of the first languages which could be used to express parallelism is Lisp, designed by McCarthy [84]. While it is unlikely that the primary intent was to support concurrent execution, functional languages have the nice property that pure functions do not have side effects and thus can be executed in parallel. More recent examples of functional languages are Haskell [57] and Erlang [6], a language created by Ericsson that is used in their communication systems. Erlang has integrated support for distributed programming. In general, it can be said that while theoretically functional languages could support parallel programming very well, in practice it has never really gained momentum.

Due to the advent of multi-core processors, the need for parallel programming languages grew, but the approaches mentioned above did not see wide acceptance. The current trend seems to be that languages that have already been successful in the sequential programming domain are extended with parallel constructs. The dominant languages include Fortran, C, C++ and Java. Automatic parallelization has been

most successful for Fortran, thanks to its stricter aliasing rules than for example C and C++. Not only compiler-based approaches have been used for Fortran. In addition, many extensions and dialects have been proposed for Fortran to support the explicit expression of parallelism. Many of these dialects were machine-dependent.

Today, more generic approaches exist. High Performance Fortran (HPF) is an extension to the regular Fortran language [3]. It provides directives, FORALL loops and restrictions in the rules for storage. Using the directives, data distribution can be defined. The FORALL construct explicitly tells that each of the iterations of a loop is independent and thus can be executed in parallel. Another extension that is available for Fortran is OpenMP [8], which is a directive based approach to specify parallelism. OpenMP is also available for C and C++, and its principles are not bound to a specific language. Co-array Fortran is an extension to Fortran 95 which is used to explicitly specify data decomposition [5, 93].

Unified Parallel C (UPC) is an extension of the C language [32], targeting both systems with a global address space and systems with disjoint address spaces. From the programmer's perspective, there is one global address space. It has a bit of an HPF flavor, in the sense that keywords are used to specify whether data is thread-local or shared. Cilk is also an extension to ANSI C introducing only three keywords: `cilk`, `spawn` and `sync`. Cilk has the property that if these keywords are removed from a Cilk program that the resulting C program is semantically equivalent to the Cilk program, if run sequentially. This simplicity is a major strength of Cilk and such simple designs will catalyze the adoption of parallel computing by the majority of programmers. Intel acquired Cilk Arts, and offers Cilk++ support.

For Java, the Titanium language offers an extension to Java for parallel execution [124]. Similar to Unified Parallel C and co-array Fortran, it offers a global memory space model on top of distributed memory architectures. Unordered loop iterations are supported (similar to FORALL loops in parallel Fortran dialects). More recently, IBM, together with academic partners, have designed the language X10 [33]. It has a Java flavor and like Titanium, it is based on the partitioned global address space principle. Its aim is to provide a scalable (on NUMA[1] platforms) solution that supports object oriented programming. Locality is expressed using *places*, such that objects and computations can be co-located.

### (Run-time) Libraries

At a higher level, parallel (run-time) libraries can provide the building blocks that a programmer can use to build parallel applications. Libraries can either support parallel execution themselves, or they facilitate the implementation of parallel algorithms. A classical library that supports the implementation of parallel applications is POSIX threads, commonly referred to as Pthreads. It provides an API that can be used to create and manage threads. Pthreads supports mutexes, condition variables and synchronization. How an application is parallelized is entirely left to the programmer. More recently, Apple released libdispatch which is a task-based system. Tasks are put in a queue and scheduled for asynchronous execution. This frees the

[1] Non-uniform memory access

programmer from worrying about thread creation. Like Pthreads, libdispatch provides an infrastructure for the explicit specification of parallelism. It does not provide parallel algorithms for any specific problem.

On a lower level, MPI (Message Passing Interface) is used [45]. MPI is one of the standard libraries used on today's supercomputing platforms, providing low-latency communication between nodes in cluster systems. Typically, MPI is used to pass data between nodes that do not share their address space. MPI does support accessing the address space of remote nodes through RDMA[2] [4]. This is not provided through memory mapped regions in the virtual memory system, but a programmer must explicitly use MPI primitives to access remote memory. GASNet takes a slightly more high-level view of parallel systems by providing an abstract parallel global address space [29]. It is used to provide a global address space for parallel languages such as UPC [32], Titanium [124] and co-array Fortran [5].

The approaches mentioned so far provide *infrastructural support* to enable the programmer to distribute computations. Another approach is to provide the programmer with an interface to enable parallel programming at the algorithmic level. STAPL (Standard Template Adaptive Parallel Library) uses this approach [31]. STAPL is an extension of C++ Standard Template Library (STL) and it provides distributed data structures and parallel algorithms. Thus, the programmer can directly express an algorithm in terms of data structures and algorithms that STAPL provide, and the STAPL run-time system will take care of distributing the different data structures and algorithms while respecting the dependencies between different tasks. Intel's Thread Building Blocks (TBB) is also a template based library aiming to express parallelism by specifying the logical parallel structure of a problem instead of explicitly writing multi-threaded software. TBB only supports shared-memory machines, whereas STAPL also supports distributed architectures.

MapReduce is a programming model for handling very large data sets [39]. Originally, it was developed at Google, but implementations for various different programming languages are available. As its name suggests, MapReduce splits computations into two parts: map and reduce. The map function basically turns input data into key value pairs. Eventually, the key value pairs produced by the map step are sorted by key, and fed into the reduce function. The reduce step can perform any operation on the values associated with a particular key. In order to gain from the MapReduce paradigm, a problem must be expressed using this formalism. This framework is especially applicable to embarrassingly parallel tasks.

## 1.3 Summary

Over the last few decades, the field of computing has made tremendous steps forward. Intel's 4004 processor consisted of 2300 transistors. Nowadays, 3 billion transistors are used in for example the NVIDIA Fermi architecture. This wealth of transistors must be put to use, and we have seen the various techniques used in processors to speed up execution. Pipelining, caches and branch prediction are all ways to speedup execution.

[2]Remote direct memory access

While processor performance has increased greatly, this does not hold for the memory system. The ever growing gap between memory performance (both bandwidth and latency) and processor performance is considered one of the major hurdles to overcome in the coming years. While task parallelism is not a new concept, the introduction of multi-core processors is a turning point in the history of computing, as it forces the wide-spread adoption of the parallel programming paradigm.

Interestingly, the idea of parallel programming has been around since the early beginnings of research in computing and as we have seen in this chapter, many approaches have been proposed to tackle the difficulties in parallel programming. At a very fine granularity, hardware solves the problem by resolving dependencies at run-time, without noticeable delay. Compilers can make these hardware extensions even more effective by selecting and ordering instructions in such a way that specific processor capabilities are exploited most efficiently. At a higher level, automatic parallelization of code has been successful, but this is mostly on regular code in which dependencies can be determined.

As automatic parallelization has seen limited success, other approaches have been taken to increase available parallelism. Support for expressing parallelism has been implemented in various programming languages. For existing, non-parallel languages, support for parallelism has mainly been added by providing software libraries to express parallelism.

One difficulty in writing parallel applications is the use of pointers and pointer-linked data structures. In the next chapter, we will treat this subject more in-depth, and outline the remainder of this thesis, in which we will focus on restructuring pointer-linked data structures such that the data layout of such structures can be altered to the actual usage pattern at run-time.

# CHAPTER 2

# Introduction

The high performance delivered by contemporary processors is made possible by an important property of the instruction streams they execute: *regularity*. High performing applications in general show regular memory access patterns. As a result, such programs exhibit high locality, thereby enabling more efficient cache usage. Regularity in the sequence of referenced memory locations is also crucial for efficient hardware-based prefetching. Predictability in branching behavior is another important factor leading to high performance. Often, regular loops execute a considerable number of iterations and only take a different branch after the last iteration. This is a perfect target for branch predictors and will result in pipelines that are fully filled most of the time. The fact that an application is regular is also visible to the compiler and regular applications are therefore relatively easy to analyze and optimize. Not very surprisingly, the list of *TOP500 Supercomputing Sites* [2] is determined using a benchmark that consists of a solver for dense linear systems, the LINPACK benchmark, which is inherently regular.

The applications described above are without doubt important, but there are many applications that do not show such regular behavior, due to a variety of reasons. For example, the hardware prefetching mechanism breaks down, if the memory access streams are not predictable. Irregularity can also be caused by dependency chains in memory, where for example a pointer chain is chased when iterating over linked lists. Some code may also show very bad branch prediction behavior.

In the previous chapter, the advances made in both hardware and software technology have been reviewed. On all fronts, at each level of granularity, attempts have been made to optimize performance and enable the definition and execution of parallel programs. For irregular problems though, the progress has been rather slow. No real, widely applicable solution has been found to this important problem. In this

chapter, we first describe the problems caused by irregularity in the context of Chapter 1. Then, work done in the area of optimizing execution of irregular applications is reviewed. Next, the general idea of the approach taken in this thesis is described, followed by a summary of the implications of this approach. Last, an outline of the remainder of this thesis is given.

## 2.1 The Problems of Irregularity

The importance of regularity for efficient execution has increased over time. In the previous chapter, we saw that in the beginning of the 1970s, the number of transistors was relatively small, no caches were used, execution was not pipelined and there was no large performance gap between the processor and the main memory. Thus, the impact of irregular memory access and constructs did not really affect performance. However, with today's complexities, such as deep pipelines, caching, branch prediction, hardware-based prefetchers and multi-core processors, this no longer holds. Any dependence or decision that cannot be properly determined or predicted by the processor will introduce delays.

From a compiler's point of view, many analyses and optimization passes fail for applications that have an irregular nature. An import cause of this failure is the presence of pointers and pointer-linked structures. As mentioned in the previous chapter, the aliasing problem plays a large role here. Parallelizing transformations can only be successful if the semantics of the execution of the applications remains the same. Pointers whose target is unknown at compile-time severely restrict the optimizations that can be applied. Not only the location of data that is pointed to affects the analysis, but also the contents of the data pointed to, as branch conditions might depend on these data. The previous chapter mentioned the polyhedral model, which is used in GCC's GRAPHITE framework. Control flow statements with data dependent conditions are a problem for most frameworks that are based on the polyhedral model. Only recently, Benabderrahmane et al. [19] have shown extensions to this model using exit predication. Still, irregular memory accesses are present and the performance may suffer. In many cases the compiler needs to be very pessimistic and it has to resort to very conservative estimates known to work in all situations.

Essentially, pointers impose the same restrictions on code analysis and optimization as code using indirection arrays. The main difference between pointers and arrays that are accessed using indirection arrays is that pointers are address space dependent, while the entries in the indirection arrays are constrained to the array bounds. As a result, automatically parallelizing computations using pointer-linked data structures on non-shared address spaces is not easily done by a compiler.

In order to circumvent the inherent limitations of automatic parallelization and data layout optimization, one can choose to either explicitly specify data structures and parallelism. While being a labor-intensive and error prone task, this approach is often taken. As described in the previous chapter, building blocks can also be provided by software libraries. The other approach mentioned was to include support for parallelism in a programming language. Such approaches imply specific choices,

which are not easily reverted, and thus might not be suitable if new technologies become available.

The problem of irregularity impacts every aspect of developing and running applications. It is not easily solved, as different data input sets will show different behavior, and will have different optimal solutions, both in terms of code and data layout. In order to solve this, both code and data must be considered by a compiler. In the next section, we will review work done in the area of data restructuring and pointer analysis.

## 2.2 Previous Work

In Chapter 1, two factors in compiler design and implementation were identified: the features of the target architecture and the available code analyses. At first, architectures were relatively straightforward and the main purpose of a compiler was to liberate the programmer from rewriting applications from scratch for each different architecture. In order to gain widespread acceptance, performance of the resulting executable program had to be reasonable. Therefore, the compiler had to exploit the architectural features, otherwise the performance would be inferior to hand-coded assembly. As described in the previous chapter, techniques like pipelining, caching, prefetching and branch prediction have made this process much more complex, but compilers are expected to keep up with all these architectural features and peculiarities.

A good example of a new architectural feature is the introduction of vector processors, for example the Cray-1 in 1976. In order to support the use of vector instructions, the compiler had to be able to identify when some operation is applied to a sequence of contiguous elements in memory. In addition, vectorization should not violate constraints implied by the program, so called data-dependencies. This requires data dependence analysis to ensure a transformation is safe [16,17,30,78,83,97,101]. Vectorizing transformations have received substantial attention [11,12,42,97,127] and most mainstream compilers, such as GCC [1] and the Intel C++ compiler [24] support vectorization. If automatic compiler-based approaches fail, one often resorts to implementing support for specific features in the programming language. Examples of this are the various Fortran dialects mentioned in the previous chapter (see Section 1.2.2).

All optimizations require dependence analysis and therefore, code that is fully analyzable at compile-time will show the best results. While optimal scheduling of instructions is NP-complete [22], efficient code can be generated, if an application can be fully analyzed. Unfortunately, many applications in the real world do not fit in this category and include many code constructs that prevent proper analysis and thus proper optimization. An important issue preventing analysis is *irregular access*. In Fortran code, this is found when arrays are accessed using index arrays, whose data is only known when the application actually runs. This naturally leads to the idea to defer some of the decisions that need to be made to run-time.

The inspector/executor technique performs optimization by splitting the execution of code into an inspector, which does run-time access pattern analysis, and an execu-

tor, which executes the optimized code generated by the inspector. Mirchandaney et al. called this a *self-scheduled approach* [87]. A few years later, Saltz, Mirchandaney and Crowley introduced the terms *inspector* and *executor* [106], which transforms the original code by first finding an appropriate schedule of so-called wave-fronts of concurrently executable loop iterations. This schedule is then used to execute the actual computations. Ujaldon et al. applied the inspector/executor paradigm to sparse matrix computations [112]. Another related approach is the hybrid analysis framework by Rus et al. [104], which provides a unified framework for both compile-time and run-time analysis. Properties that cannot be determined at compile-time will be checked at run-time.

Lin and Padua show that some indirect accesses follow particular patterns and use this information to discover parallelism [79]. They are able to analyze irregular single-indexed access, which occurs if an array is accessed using the same index variable throughout the loop, and simple indirect array access, which are accesses using an indirection array, which itself is indexed by the inner loop counter (for example, $A[B[i]]$, where $i$ is the inner loop counter).

Note that the optimizations mentioned above are necessary for multiple reasons, viewed in the context of Chapter 1. Within the processor, for example, instruction level parallelism can only be exploited if there are no dependencies between instructions and efficient use of the cache can be improved by fetching data in a particular order. From a compiler perspective, the inspector/executor paradigm is useful, as parallelism that cannot be identified at compile-time might be found at run-time. If this is successful, it simplifies the development of software considerably, shifting the burden of data segmentation from the programmer to the compiler.

The optimization of pointer-linked data structures is far more challenging than the optimization of arrays using indirect access. Pointers make it much more difficult to perform dependence analysis, as they are not constrained to specific memory regions, whereas arrays are constrained to specific regions. This lead to the development of pointer analysis, which dealt with the question whether two pointers will not, might or must point to the same location in memory [14, 55, 110]. The two most well-known pointer analyses are those of Andersen [14] and Steensgaard [110]. Pointer analyses must in general be conservative, as providing information for each possible execution path is infeasible. Algorithms must make a trade-off between performance and precision.

Aliasing is not the only problem faced when using pointers. Pointers are typically used to create recursive data structures. Such data structures can form different shapes at run-time, such as trees, acyclic graphs or cyclic graphs. Shape-analysis is concerned about determining the shape of data structures. Hummel et al. define how access patterns of data structures can be described at any point in the program [58]. Ghiya and Hendren proposed a pointer analysis that conservatively tags a heap-directed pointer as a tree, a DAG or a cyclic graph [47].

Shape information is interesting, as it provides information whether different traversal paths are disjoint and might show parallelization opportunities. Again, such analyses are essentially an extension of traditional dependence analysis. While it is of course good to know about the shape of a data structure, its conservativeness

might be a limiting factor. Therefore, next to the actual shape it is also important to recognize how data structures are actually traversed. This is recognized by Hwang and Saltz [59], who perform traversal pattern aware analysis of pointer structures. For example, a data structure might form a cyclic graph, while its traversal defines a tree.

All these analysis techniques form the basis for code and data transformations. As noted in Chapter 1, the need for such techniques has increased, due to the ever increasing complexity of processors and the enormous gap between memory performance and processor performance. Therefore, much attention has been given to reducing the effects of memory latency, a problem arising often in pointer-based applications when pointer chains are traversed. Good locality of data references is essential to reduce memory latency. Among the techniques to improve data locality are the inspector/executor strategy, which have been mentioned above.

Another approach to reduce latency and thus improve throughput is software prefetching, which is a technique that loads data from memory before it is actually needed. For pointer-linked structures, various prefetching techniques have been subject of research. Luk and Mowry [81] propose three software prefetching methods which can be applied to generic recursive data structures, namely greedy prefetching, in which pointers in a current node are prefetched, history-pointer prefetching, in which a separate history pointer is kept which stores a pointer encountered in a previous traversal, and data linearization prefetching, which reorders nodes in memory to obtain both better locality and predictability of the memory reference pattern. The data linearization technique described by Luk and Mowry is a way to enable the hardware prefetcher to be effective. Karlsson et al. [65] extend this work by combining greedy prefetching with jump pointer prefetching. They also study some applications and quantify some performance characteristics of these application, such as time spent in pointer-chasing chains and cache miss ratios. Yang and Lebeck [123] propose a hardware-assisted, active push-based prefetch mechanism. This enables the memory hierarchy to actively dereference pointers at any level in the memory hierarchy, and use this to actively prefetch data.

In the beginning of the 1990s, Gallivan et al. [46] already described the complexities of shared memories on multi-processor platforms. Though times have changed, the same problems still arise in multi-core systems. In addition, modern processors include mechanisms such as hardware prefetching and adjacent cache line prefetching, which makes the analysis of such systems even more difficult.

Memory streams have been extensively studied. For instance, Jalby et al. implemented WBTK [63], a set of micro-benchmarks to study the effect of various memory address streams on system performance. For a single-core system, address streams can in principle be obtained by getting traces. However, for multi-core systems this is different, as the processes run independently and memory references will not be handled in the same order by the memory controller when multiple applications are running simultaneously.

The potentially large impact on performance of memory intensive applications has been described by Moscibroda and Mutlu [88]. They view the multi-core problem from a security point of view. In their research, they show that concurrently running

applications can have a severe impact on each other. They call applications that slow down other applications, *memory performance hogs* (MPHs). They show that an application with a regular memory reference pattern is able to increase the execution time of an irregular application by a factor of 2.90 whereas the execution time of the regular application itself only increases by a factor of 1.18. In a simulated 16-core system they show factors of 14.6 and 4.4 respectively.

Whenever automatic transformation does not yield satisfying results, one can resort to implementing particular concepts into libraries. In order to improve locality, Rubin et al. chose to implement the concept of so called Virtual Cache Lines [103], which basically tries to store neighboring nodes in pointer-linked structures in the same cache line. While the concept is implemented as a library, the authors believe that VCL based code can be generated by a compiler. In this thesis, we do not explicitly implement VCL, but our restructuring techniques can lead to similar results. Again, it can be seen that the changes in processor technology have had their impact on every other aspect of software development, ranging from compiler-generated prefetch instructions, to software libraries taking locality into account. Closely related is the *adaptive packed-memory array*, which is proposed by Bender and Hu [20]. This is a sparse array structure which allows for efficient insertion and deletion of elements while preserving locality.

In 2001, the answer to the question posed in the title of Hind's paper [55], *"Pointer Analysis: Haven't We Solved This Problem Yet?"*, was: *No.* Today, the answer is still the same: *We have not solved this problem yet.* While all the approaches mentioned above have contributed to understanding the problems caused by pointers better, many, if not all problems stated in Hind's paper are still largely unsolved. An important point mentioned in his paper (inspired by a comment from Rakesh Ghiya) is the modeling of aggregates in weakly-typed languages and the lack of precision in such analyses.

In recent years, this topic has been researched by Lattner, who came up with practical solutions to this problem [72,73,75,76]. Rather than trying to perform shape-analysis or using type information, his work focused on determining the actual usage of data within weakly-typed languages. Their analysis (called Data Structure Analysis) is performed on an intermediate code, and thus is in principle programming language independent. This leads into a conservative segmentation in disjoint data structures, which in some cases can be proved to be type-safe. Disjoint-data structures can be pool-allocated [75] and type-safe memory-pools can be used to support structure splitting, which has been implemented for different compilers [35, 36, 48, 50]. Other (automatic) restructuring techniques that are enabled by type-safe memory pools are field reordering [34], described by Chilimbi et al., and array regrouping [126] (the inverse of structure splitting), which is described by Zhong et al. All of these techniques aim to provide more efficient use of the memory hierarchy. In this thesis, automatic structure splitting is one of the building blocks for successful restructuring of pointer-linked data structures. While Lattner's Data Structure Analysis does not solve all the problems associated with structured types (for example, inheritance hierarchies in object-oriented languages will cause the results to be very conservative),

it is a major step toward solving practical problems associated with structured types. Segmenting the data structures used in an application into disjoint regions resembles the segmentation of data structures into arrays, which is often done in Fortran code. If pointer-linked data structures can be represented as bounded arrays that are accessed using indices instead of pointers, part of the gap between pointer-based and array-based codes will be closed.

## 2.3 Our Approach

It might sound radical to say that automatic optimization of irregular code, in particular pointer-based applications, is in its infancy, but in fact no widely applicable methods are available to tackle this problem. While being a problem in single-threaded applications, this problem will be an even greater challenge for future computing platforms, for which many simultaneously running threads will be the norm. For example, the current NVIDIA Fermi platform can execute 1536 simultaneous threads. However, many restrictions apply to this *single instruction multiple threads* (SIMT) approach and transparent, concurrent execution of pointer-based code is not available. For future supercomputers, billions of threads are predicted. In order to support such architectures when using pointer-based applications, rigorous methods need to be developed. In addition, we will be facing more and more heterogeneity in systems due to the use of accelerators, mixed architectures, large scale systems with non-uniform access times and non-shared address spaces. This makes the problems inherent to the usage of pointers even larger.

If we reconsider the increasing complexity and the variety of methods that are available to harness efficient and parallel execution of software (described in Chapter 1), it is clear that no matter what solution is provided, flexibility and adaptability are key issues. Processors have different characteristics (for example, AMD and Intel processor share the same instruction set, but their implementations differ), complete systems are heterogeneous (for example, the IBM Roadrunner [70]) and input sets to problems differ. Thus, data structures should be adaptable, and architecture and address space independent. For code, the same holds. If flexibility is a requirement, code should not be specialized directly to one specific platform. Moreover, code should be adaptable, as differences in input data will give rise to different optimization opportunities.

In this thesis, we aim to lay out the foundations for a compilation environment which takes both *data layout and code* into account. Such a framework requires a unified and architecture independent representation of data structures. It must be possible to *relocate and reorder data*, according to actual access patterns emerging at run-time. A major hurdle in this process is the existence of pointers, especially in weakly-typed languages such as C. In this thesis, analysis and transformation techniques are introduced that eliminate the use of pointers in data structures and replaces pointer-based data structures by an array-based equivalent. This part of the analysis and transformation is built on top of two existing techniques, pool allocation [73, 75] and structure splitting [35, 48, 50]. We have implemented structure splitting in LLVM

and implemented a data reordering framework for (pool-allocated) pointer-linked data structures.

Using this *array-based representation*, restructuring strategies are developed, which are based on the adaptation of access patterns to conform to other access patterns present in the code. Of course, actual access patterns will not be known until run-time, but in many case, access patterns will exhibit particular features, such as injectivity (that means that with respect to a part of the iteration space, an array will only access disjoint elements). This observation allows to perform compile-time transformations into an intermediate code that is free of indirect accesses.

Data access patterns are not compile-time constants. Rather, such information becomes available when the program is run. Therefore, *access pattern aware optimizations* must be split into two phases. First, at compile-time, the code must be analyzed and instrumented to support the subsequent recompilation phase that is performed at run-time, when access patterns become available. We will develop techniques that are able to identify access patterns at run-time for pointer-linked data structures. Above, the adaptation of access patterns was mentioned. For pointer-linked structures, remapping of data is a non-trivial task. Techniques for automatic, transparent restructuring of such data structures are developed for type-unsafe environments.

Changing data layout alone can already lead to great improvements. However, there are more opportunities if code and data are optimized in concert. For example, a linked list traversal may be replaced by a loop iterating over a sequence of elements, provided that the memory is reordered such that these objects are continuous elements in memory after restructuring. We will develop methods to eliminate data dependent control flow in cases that this can be determined to be constant.

Traditionally, irregular applications are notoriously hard to optimize, as many analyses simply cannot be performed at compile-time. The approach proposed in this thesis aims to *minimize the effects of performance penalties caused by irregular properties of applications.*

The *common intermediate code* that will be obtained will allow an integrated analysis of code using arrays and code using pointer structures. As pointer-structures will be defined in terms of arrays and index arrays, the structure will not be bound anymore to a specific address space, a major limitation of today's hybrid systems.

It will be proved that even *pointer-linked data structures can be automatically restructured.* Given that this is possible, programmer-defined reorderings are not necessary anymore, and are actually even discouraged, as this complicates code. The same holds for custom memory allocators. Custom memory allocators add complexity to the application and prohibit the application of analysis and reordering transformations. This is due to the fact that memory allocators will often allocate blocks of data, and analysis techniques will not be able to distinguish between the allocation of arrays and singletons. Thus, it is preferable that the compiler and run-time system take care of data allocation and reordering instead of putting this burden on the programmer.

Next to freeing the programmer from explicitly managing memory allocation, restructuring data layout can give large performance improvements. The programmer can focus on writing the algorithm, the compiler and run-time system will adapt the

data layout according to the actual data usage. Taking this even a step further, the information on data layout can be used to further optimize code. Co-optimization of code and data can explicitly expose the actual data dependencies of a problem, eliminate data dependent loops such that a data instance specific code can be generated.

## 2.4 Outline

In this thesis, we will start with an assessment of the impact that irregularity has on a modern architecture, the Intel Core 2. In Chapter 3, a set of benchmarks, called SPARK00, is described which consists of sparse matrix codes, using orthogonally linked lists to represent its matrices. In addition, it contains several codes based on arrays only. Many of these correspond to one of the pointer-based benchmarks, such that direct comparisons between these different implementations can be made. Using the SPARK00 benchmarks, an estimate can be made of what to expect from various restructuring opportunities.

A top-down overview of the ideas behind our restructuring techniques is presented in Chapter 4. The concepts are described in an informal way, using C code samples. Safety issues that arise from the use of unsafe languages are discussed, and it is shown how an array-based representation is derived from a sparse matrix multiplication using linked lists. Using this array-based form, we show how annihilation and sublimation can be applied to this code and present results obtained using a prototype implementation of these compiler techniques.

There are many details involved in the transformation process outlined in Chapter 4. In the subsequent chapters, we take a bottom-up approach, and describe the different techniques in more detail. These chapters also include a description of the implementations of these techniques using the LLVM compiler infrastructure [74]. In order to understand the analyses and transformations that are presented, preliminary knowledge of the LLVM compiler infrastructure and Lattner's Data Structure Analysis (DSA) [72, 75] is required. Chapter 5 provides a concise overview of the background needed to understand the techniques explained in the remainder of the thesis.

Restructuring of pointer-linked structures is explained in Chapter 6. This relies on transforming pointer structures into a type-safe representation, which uses object identifiers instead of pointers. Using this position independent representation, heap data can be reordered, and both heap and stack references will be updated accordingly. In Chapter 7, a fully array-based representation is developed. This chapter also presents techniques to detect static control flow behavior at run-time, which results in optimized loop structures that have no dynamic data dependencies in their loop conditions.

The concept of sublimation is revisited in Chapter 8. It is presented in the context of a two-phase compilation process, in which sublimation is applied in the first phase to obtain a fully regular intermediate code, which is optimized into a data instance specific code at run-time, when actual access pattern become available. A case study showing the potential application of this two-phase compilation trajectory is presented

in Chapter 9, which shows how a kernel based on pointer-linked lists can be mapped to an FPGA platform, which does not have a shared memory address space with the host running the application.

We conclude this thesis with Chapter 10, which provides a retrospective view and also sheds some light on possible future directions in the field of automatic restructuring of applications using pointer-linked data structures.

## 2.5 List of Publications

Parts of this thesis have been published in journals and in conference proceedings.

**Chapter 3**

- Harmen L.A. van der Spek, Erwin M. Bakker, and Harry A.G. Wijshoff. SPARK00: A Benchmark Package for the Compiler Evaluation of Irregular/Sparse Codes. In ASCI 2008: Fourteenth Annual Conference of the Advanced School for Computing and Imaging, 2008.
- Harmen L.A. van der Spek, Erwin M. Bakker, and Harry A.G. Wijshoff. Characterizing the performance penalties induced by irregular code using pointer structures and indirection arrays on the Intel Core 2 architecture. In CF 09: Proceedings of the 6th ACM conference on Computing frontiers, pages 221–224, 2009.

**Chapter 4**

- Sven Groot, Harmen L.A. van der Spek, Erwin M. Bakker, and Harry A.G. Wijshoff. The Automatic Transformation of Linked List Data Structures. In PACT 2007: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007
- Harmen L.A. van der Spek, Sven Groot, Erwin M. Bakker, and Harry A.G. Wijshoff. A compile/run-time environment for the automatic transformation of linked list data structures. International Journal of Parallel Programming, 36(6):592–623, 2008.
- Harmen L.A. van der Spek, Erwin M. Bakker and Harry A.G. Wijshoff. Optimizing Pointer-Based Linked List Traversals Using Annihilation. Poster at ASCI 2009: Fifteenth Annual Conference of the Advanced School for Computing and Imaging, 2009.

**Chapter 6**

- Harmen L.A. van der Spek, C.W. Mattias Holm, and Harry A.G. Wijshoff. Automatic restructuring of linked data structures. In LCPC 2009: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing, pages 263–277, 2009.

**Chapter 7**

- Harmen L.A. van der Spek, C.W. Mattias Holm, and Harry A.G. Wijshoff. How to unleash array optimizations on code using recursive data structures. In ICS 2010: Proceedings of the 24th ACM International Conference on Supercomputing, pages 275–284, 2010.

**Chapter 8**

- Harmen L.A. van der Spek, H.A.G. Wijshoff. Sublimation: Expanding Data Structures to Enable Data Instance Specific Optimizations. In LCPC 2010: Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing, 2010.

# CHAPTER 3

# Characterizing the Impact of Irregularity

Measuring to what degree performance is affected by irregular behavior of an application can be done using several methods. Using hardware counters, quantities like the *instructions per cycle* (IPC), memory bandwidth, cache misses and TLB[1] misses can be obtained. In this chapter, the SPARK00 benchmarks are introduced, which consist of sparse matrix kernels and the benchmark MCF from the SPEC CPU2000 benchmarks [54]. Using the SPARK00 benchmarks, we present a systematic approach for the evaluation of performance penalties caused by irregular properties of applications. This evaluation is guided by several theses, each of which focuses on a different aspect of irregularity. These theses cover topics such as the different levels of the memory hierarchy, predictability of memory reference patterns and memory bandwidth characteristics of irregular applications. They also treat more abstract notions, such as the ability to control the impact of irregularity and the perceived irregularity of some common applications. The effectiveness of current production compilers is studied and it is shown that traditional approaches are not likely to succeed in the field of irregular applications. The pointer-based benchmarks of SPARK00 are used throughout this thesis to assess the effect of the various optimizations that target pointer-linked data structures.

## 3.1 Overview

While pointer-structured code is well known for its performance issues, we show that *pointers do not degrade performance*, but *irregularity does*. By this, we mean that if the actual emergent behavior at run-time shows regular access patterns, the code

[1] Translation lookaside buffer

that in principle contains long dependent memory access chains will perform well as hardware prefetching mechanisms will be triggered. It is also shown that the scale at which irregularity is observed has a significant impact on performance. Irregularity within a certain window (related to cache sizes) will not greatly affect performance.

It is not always possible to come up with a data layout that will result in regular behavior. There might be multiple, conflicting uses (in terms of access patterns) of a data structure. In that case, the irregularity is something one will have to accept and we determine the performance impact of not being able to predict the next address in an irregular address stream.

Memory bandwidth is one of major bottlenecks in contemporary processors. This is not only the case for regular applications, but as we show this also holds for irregular applications. Multi-core systems easily saturate the memory bus. Moreover, in irregular applications, much of the data fetched from main memory originates from wrongfully prefetched data, putting extra pressure on the memory subsystem.

The above mentioned performance penalties caused by irregular applications must be controlled as much as possible. The results of the experiments show that the problem of controlling this irregularity is not going to be solved by traditional compiler techniques, which mostly rely on optimizing control flow. It is shown, that if optimizing compilers were able to optimize data layout, this would be very effective. In other words, data driven optimization techniques show considerable performance improvements. In the remainder of this thesis, the SPARK00 benchmarks will be used to evaluate methods to automate data layout optimization (some methods even take the interaction between data and code into account).

In modern multi-core systems, the simultaneous execution of multiple processes potentially leads to quite different behavior compared to running a single process, due to resource sharing. In this chapter, the analysis on multi-core systems will be limited to the effect of multiple running processes on memory bandwidth. Multiple processes can result in an arbitrarily interleaved memory reference sequence and fine-grained analysis of such patterns is infeasible. However, at the level of memory bandwidth, specific behavioral patterns do arise and while these do not provide detailed information on the precise interaction of the different processes, it does characterize the performance limits imposed by running multiple processes on different combinations of cores.

Section 3.2 introduces the theses that characterize irregular program properties. The SPARK00 benchmarks are described in detail in Section 3.3. This section also describes the data sets used in the experiments. The experimental setup is explained in Section 3.4, including an experiment that motivates the different combinations of cores that are used in the multi-core experiments. The single-core experiments are discussed in Section 3.5 together with the theses from Section 3.2. The multi-core experiments are discussed in Section 3.6.

## 3.2 Characterizing Irregularity

The performance impact of irregularity varies widely and involves many aspects of architectures and compiler design.This section introduces our theses about irregular program properties. Theses 1 to 6 address the impact of irregularity on system performance as well as optimization constraints and opportunities. Thesis 7 focuses on the additional complexity introduced by multi-core processing.

### 3.2.1 The Impact of Irregularity on Pointer-Structured Code

Pointer-structured code is known for its unpredictable nature. This is visible at both compile-time and run-time. At compile-time, pointer-structured code will not allow for vectorized execution. Pointer-chasing loops are not very likely to generate unit stride memory accesses and data members of subsequent elements are typically not allocated sequentially in main memory which prevents vectorization. Moreover, the compiler cannot make any assumptions about the actual memory reference sequence that will occur at run-time. Pointer-structured code often contains dynamic data structures and this allows for easy insertion and deletions of new elements. While normally this causes a substantial performance decrease, we show that if the working set of a data structure is small, there is no performance degradation to be expected when memory access is irregular. Therefore, for small data sets, restructuring compiler transformations are not worth the effort and focus should be on generating optimal code. This also applies to multi-core configurations. As long as there is no competition for resources (when using small data sets), irregular memory reference streams will not affect performance.

For large data sets, for which capacity misses will occur, the situation is different. In this case, the physical data layout does matter and restructuring compiler transformations can be very beneficial.

Regarding the impact of irregular memory reference streams, we pose the following thesis:

**Thesis 1** *For applications using pointer structures, the impact of irregular memory reference streams only manifests itself when the memory size of the working set is greater than (memory) system dependent thresholds, such as L1 and L2 cache sizes.*

### 3.2.2 The Predictability of Memory Reference Streams

Regular applications have the advantage that they can be analyzed relatively easily at compile-time. This also implies that their memory reference patterns are in principle predictable. For irregular applications this is different. For pointer-chasing loops, the memory reference sequence that is generated can be anything. In this chapter, we will show the performance impact of being able to predict the next memory reference in a pointer-chasing loop. We also measure the performance penalty of not being able to predict future references in the case memory is accessed randomly by a pointer-chasing loop. We do so by replacing the pointer chasing loop by a loop that breaks the cross-iteration dependency.

Codes that use indirection arrays to implement dynamic data structures have an advantage over pointer-based codes. In indirection array based codes, data is stored in disjoint arrays and therefore cache utilization is likely to be better than in the case where structure nodes are used. When using structure nodes, it is more likely that unneeded data members are fetched into the caches. While these factors can have a significant impact, we will show that they have less influence on performance than the unpredictability of memory reference streams.

When running multiple concurrent processes, unpredictability in memory reference patterns is a more complicated problem. Instead of handling a single address stream, different address streams originating from different processes are interleaved, introducing an extra degree of complexity.

These considerations lead to the following thesis:

**Thesis 2** *The main performance loss in irregular code is caused by unpredictability of memory reference streams rather than using pointer-structured code vs. code using indirection arrays.*

### 3.2.3 Memory Bandwidth in Irregular Applications

If an application emits a very irregular memory stream in which subsequent memory references are depending on each other, then it suffers from high latency in the memory subsystem. In this case, the full memory bandwidth is not utilized. For multi-core configurations, memory bandwidth is a fundamental constraining factor. This is described more in depth in Section 3.2.7.

While much of the impact of irregularity can be controlled by choosing an appropriate data layout, there are cases where this does not solve the problem. For example, if there exist multiple access patterns for which no proper common data layout can be used. Such patterns are common in for instance search trees which are traversed differently for different search keys. In this case, we suggest that the unused memory bandwidth is put to use. This is formulated in the following thesis:

**Thesis 3** *Irregular applications waste memory bandwidth. This extra bandwidth should be used by these applications to provide extra information on memory reference patterns, to make those patterns predictable. See also [65, 81].*

### 3.2.4 Controlling the Impact of Irregularity

Code which appears irregular, can behave quite regularly. In our experiments, we have found that if a data layout is chosen carefully, performance improves substantially as the irregular code issues a more or less regular memory reference sequence. The impact of irregularity on the compiler is more difficult to deal with and this is also reflected in Thesis 6, where it is stated that current compiler technology does not suffice to deal with irregularity.

In a multi-core setting, this problem of conflicting data layouts also exists. Unintentionally, different data structures might cause false sharing of cache space. Such interactions between multiple processes is not taken into account by current compilers.

Another approach is needed, and we will show that compilers should shift their focus to data restructuring. We summarize this with the following thesis:

**Thesis 4** *Much of the impact of irregularity can be controlled by choosing an appropriate data layout.*

### 3.2.5 Irregularity of Sparse Code

Contrary to common belief, sparse code is not necessarily very irregular. Irregularity is a concept that is hard to grasp, and even dependent on for instance the platform an application runs on. For example, if the only irregular access in a loop is access to a vector that fits in the cache, and the architecture does not support vectorized execution, then the impact of this irregular access is negligible. However, if vectorized execution is available, the irregularity has a much higher impact, as the indirect array access prevents vectorization. But sparse matrix algorithms do not necessarily access data in an irregular fashion, as is seen in for instance one of the benchmark kernels, IJACIT:

```
for( ; j <= j2; j++ ) {
    x_2[i] -= a[j] * x_1[ ja[j] ];
```

In this case, the only irregular access is to $x\_1$, which depending on the size of the array has more or less impact. However, the rest of the loop shows regular behavior, and apart from the consequences of not being able to vectorize, data access is just as efficient as on dense code. A difference between sparse and dense code is that in sparse code, more data must be fetched from memory, i.e. the index array. As this is also a regular memory reference stream, it can be expected that the memory subsystem will be used efficiently, and the performance penalty in such cases is not expected to be that great.

The following thesis concerns the often perceived irregularity of sparse matrix algorithms:

**Thesis 5** *Sparse matrix algorithms are often regarded as very irregular applications. This is exaggerated. In fact, especially for codes using indirection arrays, the actual behavior is close to implementations operating on dense matrices.*

### 3.2.6 Optimizing Compilers

Optimizing compilers have been very successful. Not only do they provide platform independence, but they have also shown to be able to generate highly efficient code, in many cases even more efficient code than hand-written assembly. Most of these successful optimization target regular loops and much work on loop reordering transformations has been done on code containing loops with regularly accessed data. For irregular codes, however, many such compiler techniques cannot be applied. Pointers may be aliased, the expression used to index an array is data dependent and loop bounds cannot always be determined at compile-time.

Some compilers perform a program-wide analysis. This is a desirable property as a non-global view cannot provide a complete view on the use of data structures and will prevent any form of data restructuring, especially in type-unsafe languages such as C. However, such analyses do not take into account the interactions between multiple processes. This is a subject that must be addressed in future compilers as multi-core platform are becoming the norm.

The Intel C Compiler, which has been used in the experiments, provides various optimization levels, namely O1, O2 and O3. The benchmarks have been compiled and run using all three of these options in order to compare their relative performance on both regular and irregular code.

In this area, there is much to be gained, as often at run-time there are no actual data dependencies. However, current compiler optimizations that mostly target code only without considering data are not yielding satisfactory results on irregular code. This is formulated in the following thesis:

**Thesis 6** *Code and control flow optimizations as done by current optimizing compilers have little impact on the overall performance of irregular codes, now, and in the near future.*

### 3.2.7 Irregularity in Multi-Core Environments

The theses above do, in principle, also apply to the individually running processes in a multi-core environment. However, the interaction between those concurrently running processes adds another dimension to the problem. This consists of the sharing of resources, more specifically the caches and the memory bus. We have mentioned some characteristics of multi-core platforms above.

The main difference between running a single process and running multiple processes is the emerging interleaved memory reference patterns. Whereas in a single memory reference stream a pattern might be identified, this becomes less likely if multiple address streams are merged at the memory controller. Also, if all separate memory reference streams are irregular, it is possible that memory bandwidth is saturated and becomes a problem.

An important difference between single and multi-core configurations is summarized in the following thesis:

**Thesis 7** *In multi-core environments memory bandwidth is the main constraining factor, even if applications show irregular behavior.*

## 3.3 The SPARK00 Benchmarks

The SPARK00 benchmarks are used to measure the performance of irregular codes. It is split into three groups of benchmarks: the first group implements applications that are based on pointer structures. The irregularity in these applications is due to the traversal of pointer chains and arrays that are accessed using input data-dependent expressions. The second group consists of applications that are based on indirection arrays.

The main difference between these groups is that the pointer-structured codes group data into nodes, whereas in the array based codes, the same data members are in the same array. The third group implements dense counterparts for some sparse kernels, for comparison purposes.

The pointer-based sparse matrix benchmarks are implemented using orthogonally linked lists. The design is based on the SPARSE library [69], but SPARK00 uses a much smaller memory footprint by removing all metadata from the structure representing matrix elements. In addition, SPARK00 does not use a custom memory allocator, as the optimization of memory allocation and data layout should be optimized automatically, and SPARK00 leaves the challenge to the compiler.

MCF is a pointer-based benchmark from the SPEC CPU2000 benchmark suite [54, 80] The benchmarks using indirection arrays are partially based on code from Saad and Wijshoff's SPARK [105] benchmark.

### 3.3.1 Description of the Benchmarks

The first subset of benchmarks is the pointer-based part of SPARK00, which consists of the following programs:

1. *SPMATVEC. Sparse matrix times dense vector.* The sparse matrix is represented using compressed row storage. The rows themselves are stored using linked lists. Each row is traversed and each element is multiplied with the corresponding element in the dense vector. The pointer traversal is one cause for irregularity. The other cause of irregularity is the indexing of the dense vector by a structure member of the linked list nodes (the column index of the sparse matrix element is used to index the vector). The result is stored in a separate dense vector.

2. *SPMATMAT. Sparse matrix times dense matrix.* The sparse matrix is represented using compressed row storage, the same manner as in SPMATVEC. The dense matrix is a C-style 2-dimensional array, which is dynamically allocated. This is different from FORTRAN-style 2-dimensional arrays, in which a contiguous block is accessed by an affine function of the loop index variables. Therefore, to access an element, two indirections are needed to access the appropriate value in C. The main difference with SPMATVEC is that in this benchmark, values are indirectly accessed whereas in the other benchmark, pointers are indirectly accessed.

3. *JACIT. Jacobi iteration.* Jacobi iteration [49] is used to solve $Ax = b$. The sparse matrix $A$ is represented using linked lists, compressed row storage. The linked list is traversed using two subsequent loops. One loop handles the elements before the diagonal and the second handles the elements after the diagonal. This traversal which is spread over two *while*-loops involves a termination condition in the first *while*-loop which is input data dependent.

4. *DSOLVE. Solve a linear system $Ax = b$ using forward substitution and backward substitution.* The matrix is represented using orthogonal linked lists (the matrix

is traversed both row-wise and column-wise). The procedure takes a matrix that has been LU-factorized and solves $Ax = b$. In order to do this, the right hand side vector is permuted into an intermediate vector, after which forward substitution is applied to solve $Lc = b$. The forward substitution traverses the matrix column-wise. Next, $Uc = x$ can be solved by backward substitution which traverses the matrix row-wise. Finally, $x$ is permuted to obtain the result in the desired order.

5. *PCG. Preconditioned conjugate gradient.* PCG iteratively solves $Ax = b$. It uses the compressed row storage scheme implemented using linked lists. The code features indirect access caused by pointer traversals, as well as an array indexed by structure members of the list nodes. This array is also used in a regular fashion. Although the main computational part of PCG is the same as SPMATVEC, PCG uses the outcome of the multiplication in subsequent dot product operations.

6. *MCF. Minimum cost flow problem solver.* 181.mcf [80] is a program from the SPEC CPU2000 [54] benchmark suite that solves the minimum cost flow problem. The network simplex implementation is a pointer intensive application that is known to exhibit very poor cache performance due to the irregular nature of the memory access patterns caused by extensive use of pointer-linked data structures.

The second subset consists of codes in which the irregular access originates from the use of indirection arrays. ASM, TRMAT, CMcK and MPERM are taken from the SPARK benchmark suite [105] and have been translated to C. ISPMATVEC, ISPMATVEC, ISPMATMAT and IJACIT are the indirection array-based counterparts of their point-based equivalents. IJDSPMATVEC is a jagged diagonal implementation of sparse matrix vector multiplication.

1. *ISPMATVEC. Sparse matrix times dense vector.* Contrary to its pointer-based version (SPMATVEC), this version uses the commonly used compressed row storage format (CRS). This consists of an array storing the sparse matrix data by row, an array storing the offset at which each row start, and an array storing the column index of each matrix element.

2. *ISPMATMAT. Sparse matrix times dense matrix.* The array-based counterpart of SPMATMAT.

3. *IJACIT. Jacobi iteration.* The array-based counterpart of JACIT.

4. *IJDSPMATVEC. Sparse matrix times dense matrix.* Array-based version of sparse matrix times dense vector, using the jagged diagonal storage format.

5. *ASM. Assemble stiffness matrix.* Finite element methods involve an assembly step, in which all interactions between sub-elements consisting of 3-node triangular element are merged into one global matrix. Access to this matrix is governed by the connectivity matrix which is used to index the global array.

The input data set used for this benchmark is the *wrench* data set, which is depicted in Figure 3.1.

6. *TRMAT. Transpose a matrix.* Computing the transpose of a sparse matrix contains quite some irregularity. First of all, the number of elements in a column is not known beforehand, and a traversal of the old index structure is needed to accumulate the right number of elements per column. This results in many scattered updates. The column counts are then translated into array offsets, which is done by a regular loop with one read-after-write dependency. Next, all data and index elements from the original matrix are traversed and mapped to the corresponding locations in the new arrays, causing single and double indirect access to arrays. The vector containing the row offsets is used to remember the current row offset within the target matrix. As a result, all elements of this vector must be moved one position to the right after filling the column and data vectors.

7. *CMcK. Compute Cuthill-McKee ordering.* The Cuthill-McKee method [37] computes a permutation array that aims to reduce the bandwidth of a sparse matrix. It does so by interpreting the sparse matrix as an adjacency matrix and computes a relabeling of the nodes. The relabeling is computed as follows. A breadth-first search is started at the node within minimal degree, which is labeled 1. Next, all adjacent nodes are considered and relabeled, starting with the node with lowest degree. The relabeling is recorded in the permutation array. The newly labeled nodes are expanded (following the ordering defined by the new labeling) and all unlabeled nodes are relabeled. This process continues until the entire connected component to which the starting node belongs is relabeled. If there still are nodes left, a remaining node with minimum degree is picked and the process above is repeated, until all nodes have been relabeled. The irregularity stems from the permutation array and the array that stores the column indices. The permutation array is used to locate the nodes that must be traversed next during the breadth-first search. Loop bounds and conditional branches are data dependent, which further complicates analysis.

8. *MPERM. Perform a symmetric permutation* $B = PAP^T$ of an array $A$ and its associated right hand side vector $b$. where $P$ is the permutation matrix. Instead of storing the permutation matrix, the mapping is stored in an array. Irregularity occurs naturally in permutation problems. The permutation requires a complete scan of the row index array to determine the new row sizes. This traversal mixes both regular and irregular access. Using the new row sizes, the new offsets are computed. Next, the iteration space of the newly generated index structure is traversed and the corresponding data from the original data structure is copied, which involves indirect accesses.

The third group consists of dense implementations of some of the sparse benchmarks, that closely correspond to their sparse counterpart. For the following benchmarks, dense versions have been implemented: SPMATVEC, SPMATMAT, JACIT,

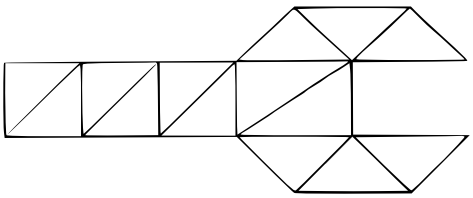


Figure 3.1: The wrench data set used in the ASM benchmark.

DSOLVE and PCG. They are referred to as DMATVEC, DMATMAT, DJACIT, DDSOLVE and DPCG respectively.

For multi-core experiments we focus on benchmarks that either stress the memory bandwidth or exhibit irregular behavior because of pointer usage. We have selected the following benchmarks as a representative subset: SPMATVEC, PCG, MPERM and JACIT. According to additional experiments conducted, the other benchmarks show similar behavior with respect to bandwidth utilization and its associated performance degradation.

### 3.3.2 The Input Data

The input data sets for the current release of SPARK00 have been selected from the University of Florida Sparse Matrix Collection, which is maintained by Davis [38]. The matrices have been selected from different problem domains. Matrices from a wide range of sizes (measured in number of non-zero elements ($nnz$)) have been chosen to compare the behavior of applications when data resides in different levels of the memory hierarchy. Table 3.1 gives a summary of the characteristics of the matrices used in SPARK00. The number of non-zero elements, prefixed with 'M' is used as identifier for the matrices.

For the benchmarks SPMATVEC, ISPMATVEC, IJDSPMATVEC, SPMATMAT and TRMAT, all matrices are used. For CMCK and MPERM, only symmetric matrices are used. For the solvers JACIT, PCG and DSOLVE, only matrices with full diagonals are used. As DSOLVE performs a LU-factorization prior to solving $Ax = b$, elements are potentially added to the matrix due to fill-in. Therefore, only matrices that after factorization do not exceed the size of the largest matrix from the initial input set are used.

For each sparse matrix, a dense matrix is generated with approximately the same number of non-zero elements. These matrices are prefixed with 'D' (for example, D100 is a dense 10x10 matrix). These matrices are used to compare sparse algorithms with

| ID | Matrix | Size | NNZ LU | Symmetric | Problem Domain |
|---|---|---|---|---|---|
| M82 | Oberwolfach/LF10 | 18 | 98 | Yes | Model reduction |
| M271 | HB/impcol_b | 59 | 924 | No | Chem. proc. simulation |
| M479 | Bai/rw136 | 136 | 2318 | No | Statistical/math. problem |
| M665 | Rajat/rajat11 | 135 | 920 | No | Circuit simulation |
| M1083 | HB/bcsstm09 | 1083 | 1083 | Yes | Structural problem |
| M1614 | Sandia/oscil_trans_01 | 430 | 2704 | No | Circuit simulation |
| M2474 | HB/662_bus | 662 | 4572 | Yes | Power network |
| M2580 | Bai/rdb450l | 450 | 12082 | No | Comp. fluid dynamics |
| M3068 | HB/str_200 | 363 | 6439 | No | Subsequent optimization |
| M7419 | Norris/lung1 | 1650 | 7419 | No | Comp. fluid dynamics |
| M24270 | Boeing/bcsstm34 | 588 | 87742 | Yes | Structural problem |
| M41594 | vanHeukelum/cage9 | 3534 | 1260004 | No | Directed weighted graph |
| M61896 | Zitney/rdist3a | 2398 | 1641781 | No | Chem. proc. simulation |
| M72734 | Hollinger/jan99jac040 | 13694 | 546686 | No | Economic |
| M105339 | Boeing/crystm01 | 4875 | 707199 | Yes | Materials |
| M578890 | Sandia/ASIC_100ks | 99190 | 3894306 | No | Circuit simulation |
| M680341 | Norris/heart3 | 2339 | 1315281 | No | 2D/3D problem |
| M713907 | VanVelzen/Zd_Jac3_db | 22835 | NA | No | Chem. proc. simulation |
| M715804 | ACUSIM/Pres_Poisson | 14822 | 5376356 | Yes | Comp. fluid dynamics |
| M726674 | AMD/G2_circuit | 150102 | NA | Yes | Circuit simulation |
| M1143140 | Boeing/bcsstk36 | 23052 | NA | Yes | Structural problem |
| M3279690 | ND/nd3k | 9000 | NA | Yes | 2D/3D problem |

Table 3.1: The matrices in this table are used in the experiments. They are listed by increasing size (number of non-zero elements). The number in the ID column shows the number of non-zero elements. Size is the numbers of rows and columns (all matrices are square). NNZ LU denotes the number of non-zero elements after LU factorization (NA indicates NNZ LU exceeds the size of the largest matrix (non-factorized) in use, M3279690.

their dense counterparts.

MCF uses the data sets from the SPEC2000 benchmark suite. This are the *test*, *train* and *ref* data sets. These sets differ in their problem sizes. The ASM kernel uses the *wrench* data input set as described in [107]. Different input data sets are generated by specifying a different number of initial grid points and a varying number of subsequent mesh refinements. For MPERM, the permutation matrix which results from the CMCK benchmark is taken and applied to the corresponding matrix.

In the multi-core experiments, many different core configurations are tested. Therefore, a number of representative matrices have been selected for these experiments. The following matrices are used: $M82$ (*LF10*), $M271$ (*impco_b*), $M24270$ (*bcsstm34*), $M41594$ (*cage9*), $M726674$ (*G2_circuit*) and $M3279690$ (*nd3k*). For the benchmark SPMATVEC, only the data set *nd3k* is used. For CMCK, only the symmetric matrices from this set are used. The permutation produced by CMCK is used as the permutation vector in MPERM. JACIT and PCG use all matrices except *LF10*, which contains zero elements on the diagonal.

## 3.4 Experimental Setup

### 3.4.1 Hardware and Software Configuration

The benchmarks have been executed on an Intel Core 2 based Intel Xeon E5420 running at 2.50GHz in 64-bit mode The system contains 8 cores (2 chips, each containing 4 cores) and has a total of 32GB of physical memory. The FSB clock speed is 1333MHz. Figure 3.2 depicts the layout of the CPUs, as reported by the Linux

kernel (using *"cat /proc/cpuinfo"*). The compiler used to compile the benchmarks is the Intel C compiler, version 10.1. The benchmarks are compiled with the options '-O2'. The optimization level '-O3' is not used, because this applies more aggressive optimizations that in general are not effective on irregular code. This can even slow down code [117].

The benchmarks used can be configured to use the Intel Performance Tuning Utility (PTU) [62] to obtain performance data from the hardware performance counters. PTU uses statistical based event sampling to monitor for specific events that can be specified by the user. In the experiments presented here, the following events have been monitored for the single core experiments:

- BUS_DRDY_CLOCKS.ALL_AGENTS
- BUS_TRANS_ANY.ALL_AGENTS
- CPU_CLK_UNHALTED.BUS
- CPU_CLK_UNHALTED.CORE
- INST_RETIRED.ANY
- L2_LINES_IN.SELF.ANY
- L2_LINES_IN.SELF.DEMAND

Using these events, the ratios *Clocks per Instructions Retired (CPI)*, *L2 Cache Miss Rate*, *L2 Cache Demand Mis Rate*, *Bus Utilization* and *Data Bus Utilization* can be computed.

In order to run the benchmark kernels concurrently, a synchronization point is inserted before the kernel entry point, such that all instances will start execution of the benchmark kernels simultaneously and the results are not distorted by IO operations from other instances. This synchronization is done using System V IPC semaphores.

### 3.4.2 Data Layout

The impact of irregularity is assessed by generating different data layouts. For the single-core benchmarks, three different input sets are generated for each matrix, which are ordered differently. This different order is reflected in the pointer structures that are built (different order of insertion). The three input formats are: the *CSR format*, in which the entries are ordered row-wise, the *CSC format*, in which the entries are ordered column-wise and the *RND format*, in which the entries are ordered randomly.

For the pointer-based benchmarks, the allocation of memory is done using large blocks, such that most elements that are inserted after each other will also be subsequent elements in main memory. Using these different memory layouts, both regular and irregular memory reference streams are generated by the irregular pointer applications. The results are used to determine the impact of the reference patterns at the various memory levels. The data layouts used are named after their respective

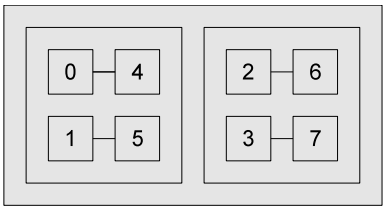



Figure 3.2: Core layout as reported by "*cat /proc/cpuinfo*". The system features 2 chips, each featuring 4 cores. The cores are numbered as reported by the Linux kernel. Cores that are connected in this figure share resources which has been determined experimentally. This is shown in Figure 3.3.

input data sets, CSR, CSC and RND. Note that the indirection array based codes use the CSR input sets, and their internal storage format is also called CSR (Compressed Sparse Row), except for IJDSPMATVEC, which uses JDS (Jagged Diagonal Storage).

In the multi-core benchmarks only the CSR and RND formats are used. They are referred to as 'S' (sequential) and 'R' (random) in the figures.

### 3.4.3 Selection of Core Combinations for Multi-Core Experiments

As there are many combinations of cores to choose from when running an on 8 core system (256), we only use combinations consisting of 1, 2, 4 or 8 cores. To further restrict the number of combinations, a set of representative combinations has been determined. This is done by determining the *clocks per instruction retired* (CPI) for the benchmark SPMATVEC, using many different combinations. The largest data set (*nd3k*) is used with the sequential input order, such that the generated memory reference sequences by the pointer traversals will be sequential. This data set does not fit in the L2 caches. The results are shown in Figure 3.3.

For 2 cores, three levels of performance exist. The fastest level (lowest CPI) is obtained by running on 2 cores that are on a different chip. The slowest level is obtained by 2 cores running on the same chip. Most likely, these two cores share resources, like the L2 cache. This combination will be called the *slow* combination throughout this chapter. The intermediate level is obtained when 2 cores are on the same chip but do not obtain the worst performance. This configuration is referred to as the *fast* combination. For 4 cores, some patterns can be observed. Fastest performance is obtained by picking those 4 cores that consist of two pairs that run in the *fast* configuration as defined above. As expected from the results on 2 cores, the configuration in which 4 cores are selected from 1 chip performs worst. The configurations with the largest spread in performance occur when 3 cores are selected from 1 chip, and the remaining core from the other chip. This configuration (#10 in Table 3.2) is not used in further experiments, as we chose only to run identical processes that are equally distributed among the two chips. The configurations where

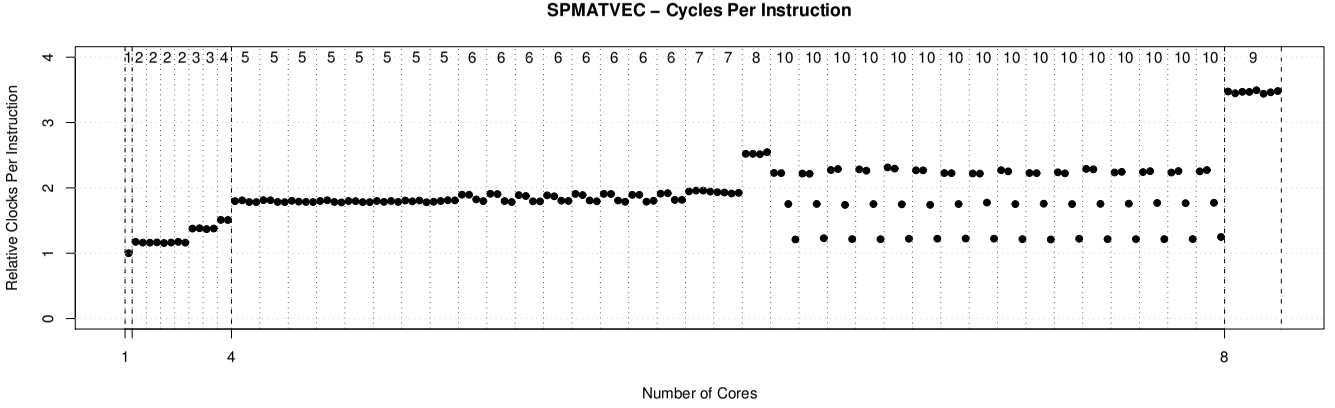



Figure 3.3: SPMATVEC - Performance in clocks per instruction (CPI) using the data set *nd3k* for various combinations of 1, 2, 4 and 8 cores. Different configurations are separated by vertical dotted lines. Each data point depicts a process. Each column is annotated with a number, which indicates the configuration type as given in Table 3.2.

there are two distinct groups with slightly different performance are the configurations where 2 pairs are chosen from different chips, such that 1 pair is running in a *slow* combination and 1 pair in the *fast* combination. The remaining configurations consists of either 2 pairs running both in a *fast* combination or in a *slow* combination.

The results show that running multiple instances concurrently has a significant impact on the performance of the different processes. The different configurations show clear patterns and these patterns have been used to select a representative subset of configurations that will be used in the remainder of this chapter. Table 3.2 summarizes the representative combinations of multiple cores that will be used throughout the remainder of this chapter.

## 3.5 Experiments on a Single Core

### 3.5.1 The Impact of Irregularity on Pointer-Structured Code

In order to verify Thesis 1, which states that irregularity in pointer-structured code only causes performance degradation in the case that the working set size exceeds particular (memory) system limits, we will isolate the effects of irregularity in memory accesses. Therefore, we do not specifically look at the performance for different data sets, in which larger data sets will in general perform worse than smaller data sets, but instead we will focus on the effects of irregularity in memory reference streams for different data set sizes.

The evaluation of different memory access streams is realized by running the same program with a different memory layout. This is possible for the benchmarks which use pointer structures to represent the matrices. By controlling the order in which elements are added to the matrix, different memory access patterns emerge when traversing the pointer structures, showing different behavior. Here, the three memory layouts described in the beginning of this section are used (CSR, CSC and RND).

Figure 3.4 shows the relative CPI of the benchmark DSOLVE. For each data set, the three different memory layouts are used and the resulting relative CPI is obtained by using the RND sets as the reference CPI. The matrices are ordered from left to right by increasing number of non-zero elements (after LU-factorization). DSOLVE is quite interesting, as the results deviate from the results obtained on the other benchmarks. In DSOLVE, the different memory layouts do not seem to be very influential. This is because the structures are built prior to LU-factorization, during which zero elements become non-zero elements (fill-in) and new nodes are inserted in the matrix. Eventually, this results in a data structure whose associated memory reference stream upon traversal behaves very similar to a random sequence. Figure 3.5 shows the relative CPI for PCG. In this case, the memory layout is not changed prior to running the benchmark and the performance of the memory layouts resulting from row-wise and column-wise insertion order (CSR and CSC) show a much lower CPI. The results of PCG are also typical for the benchmarks SPMATVEC, SPMATMAT and JACIT.

The pattern observed in Figure 3.5 indicates that for small data sets, the irreg-

| Combination ID | Selected cores | Description |
|---|---|---|
| 1 | 0 | Single-core |
| 2 | 0; 2 | 2 cores, each on a different chip |
| 3 | 0,1 | 2 cores, both on the same chip, fast combination |
| 4 | 0-4 | 2 cores, both on the same chip, slow combination |
| 5 | 0,5; 2,7 | 4 cores, both pairs 0,5 and 2,7 are a fast combination |
| 6 | 0-4; 2,7 | 4 cores, pair 0-4 is a slow combination, 2,7 a fast combination |
| 7 | 0-4; 2-6 | 4 cores, both pairs 0-4 and 2-6 are a slow combination |
| 8 | 0-4, 1-5 | 4 cores, all on the same chip |
| 9 | 0-4, 1-5; 2-6, 3-7 | 8 cores |
| *10* | *0-4, 1; 2* | *4 cores, 3 on one chip, 1 on the other* |

Table 3.2: Core combinations used in the experiments. These are determined from the experiment shown in Figure 3.3. Slow combinations are separated by '-', fast combinations by ',' and cores on different chips are separated using ';'. The asymmetric configuration 10 is not used in the further experiments.

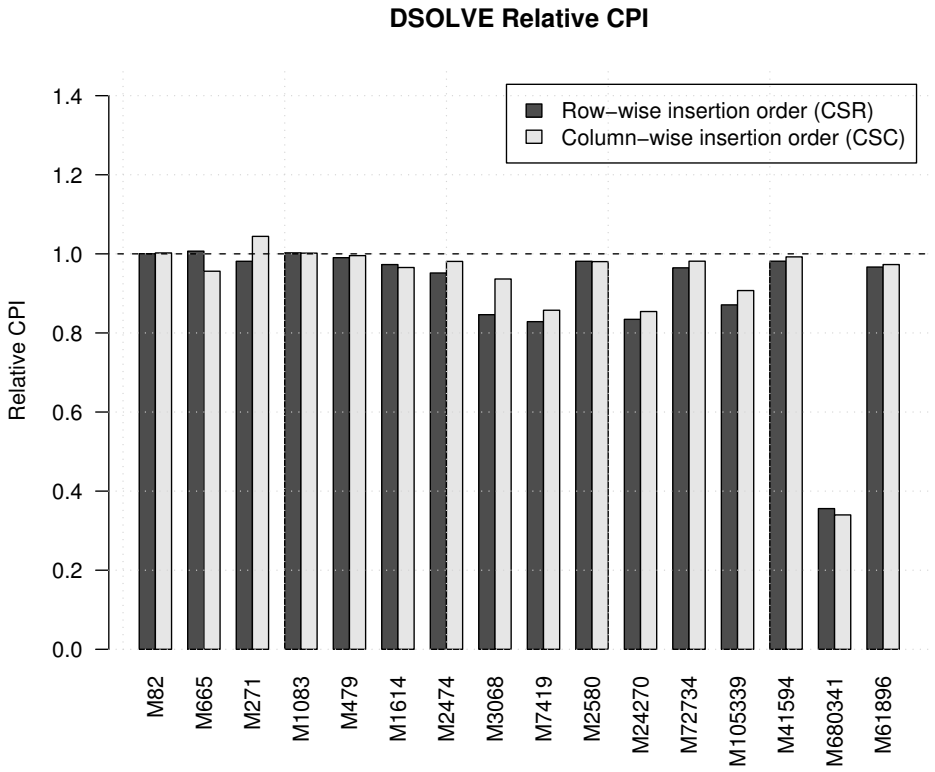



Figure 3.4: DSOLVE - Cycles Per Instruction (CPI), Relative to RND data set. The benchmarks have been run using three different memory layouts resulting from different insertion orders (CSR, CSC and RND). The CPI has been normalized by the CPI obtained using the RND data set (random insertion order). On the x-axis are the different input data sets (See Table 3.1) ordered from left to right by increasing size. The results for PCG are shown in Figure 3.5. The improvements by choosing a regular initial layout does not have great influence on DSOLVE in most cases, as irregularity in the data structure is introduced by the LU factorization done prior to running the actual benchmark.

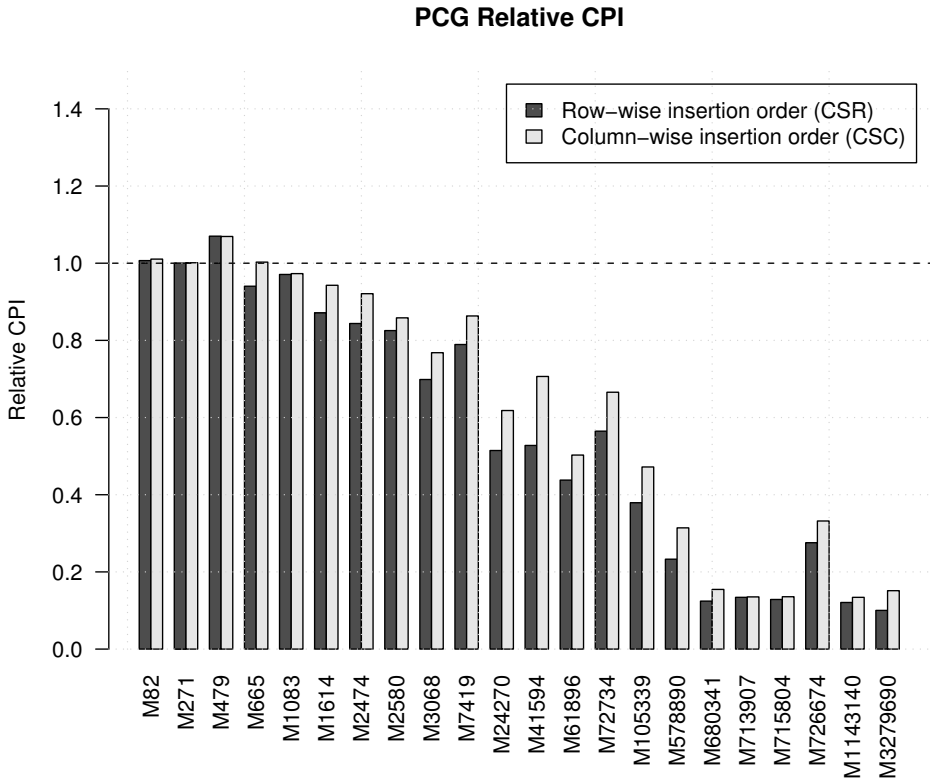



Figure 3.5: PCG - Cycles Per Instruction (CPI), Relative to RND data set. PCG shows behavior that is typical for the different memory layouts. For the small data sets, the data layout has no influence. The larger the data set becomes, the largest the impact of the data layout chosen. SPMATVEC, SPMATMAT and JACIT show similar behavior.

ularity in the memory reference stream does not affect performance in most cases. Whenever the input data sets grow larger, the difference between the different memory layouts becomes apparent, and the two more regularly ordered memory layouts clearly show better performance than the randomly ordered data structure. In the other benchmarks, these performance characteristics are also observed for the sparse matrices. Note that, while CSR performs best in most cases, CSC, which does not result in a perfectly regular address stream, performs relatively well and it can be concluded that perfect regularity in a memory reference stream is not strictly necessary to obtain good performance. Table 3.3 summarizes the range of relative CPIs measured for the CSR and CSC layouts, with respect to the RND layout. This has been done for all pointer-based matrix benchmarks.

The experiments support Thesis 1 that the impact of irregularity only manifests itself in pointer-based applications, if large enough data sets are used. The thresholds at which the impact grows are related to the respective sizes of the L1 and L2 cache. This property should be used in the trade-off whether one should restructure data or not. For small data sets, this is not necessary as it will not yield any performance gains. However, the larger the data set, the greater the potential improvements that can be obtained by restructuring data.

### 3.5.2 The Predictability of Memory Reference Streams

Thesis 2 states that the real reason of performance degradation is not really a fundamental problem of pointer-based code, but that the real problem is the unpredictability of memory reference streams. It is important that cross-iteration dependency chains do affect performance as least as possible. In SPARK00, the different memory layouts do have a severe impact on performance, which has been shown in Section 3.5.1, and this can be partly traced back to the cross-iteration dependency of the pointer traversal. To quantify the performance degradation caused by this dependency, a test program is used, which is based on SPMATVEC. Instead of traversing the matrix elements by a linked list traversal, the pointers to the nodes are stored in an array. This way, the memory reference stream of pointer nodes can be determined by the processor without being hampered by the cross iteration data dependency, allowing for a rough assessment of the effectiveness of the prefetching mechanism. The original inner loop looks as follows:

```
while( pElement ) {
    result[row] += pElement->Real * right[pElement->Col];
    pElement = pElement->NextInRow;
}
```

The modified loop which uses arrays to fetch the pointers to the matrix elements becomes:

| Benchmark | Rel. CPI | | Abs. L2 Miss Rate | | |
|---|---|---|---|---|---|
| | CSR | CSC | CSR | CSC | RND |
| SPMATVEC | 0.10-1.04 | 0.12-1.03 | 0.00-0.08 | 0.00-0.09 | 0.00-0.59 |
| SPMATMAT | 0.37-0.98 | 0.42-1.00 | 0.00-0.01 | 0.00-0.01 | 0.00-0.02 |
| JACIT | 0.10-1.00 | 0.13-1.00 | 0.00-0.07 | 0.00-0.08 | 0.00-0.48 |
| DSOLVE | 0.36-1.01 | 0.34-1.04 | 0.00-0.28 | 0.00-0.28 | 0.00-0.31 |
| PCG | 0.10-1.07 | 0.13-1.07 | 0.00-0.06 | 0.00-0.07 | 0.00-0.46 |

| Benchmark | | | Abs. Data Bus Utilization | | |
|---|---|---|---|---|---|
| | | | CSR | CSC | RND |
| SPMATVEC | | | 0.00-0.27 | 0.00-0.26 | 0.00-0.19 |
| SPMATMAT | | | 0.00-0.28 | 0.00-0.25 | 0.00-0.22 |
| JACIT | | | 0.00-0.27 | 0.00-0.25 | 0.00-0.19 |
| DSOLVE | | | 0.00-0.19 | 0.00-0.19 | 0.00-0.19 |
| PCG | | | 0.00-0.33 | 0.00-0.31 | 0.00-0.29 |

Table 3.3: Summary of pointer-based benchmarks. The table lists relative CPI (cycles per instruction, relative to the RND data set), absolute L2 cache miss rate and absolute data bus utilization.

```
j = 0; pElement = pRow[j];
while( pElement ) {
    result[row] += pElement->Real * right[pElement->Col];
    j++;
    pElement = pRow[j];
}
```

In this loop, *pRow* is a pointer to the array of pointers for the current row. This loop shows largely the same behavior as the loop above, except that pElement can be determined independently for each iteration. The extra array will put some extra load on the memory subsystem, but as the maximum data bus utilization observed for SPMATVEC has been 0.2743, there is enough bandwidth available to accommodate for this extra data (the maximum bandwidth has been determined to be 0.4746, see Section 3.5.3).

Figure 3.6 shows the normalized execution times of SPMATVEC using the prefetched pointer arrays. The reference execution time is the original version of SPMATVEC using the same data set with the same memory layout. Execution time is used in this case and not CPI, because both kernels differ at the assembly level. The figure shows that whenever the hardware prefetching mechanism is effective, which is true for the CSR and CSC data layouts, then breaking the dependency chains can both have a positive or negative effect, depending on the input data. However, these effects are relatively small, compared to the effect that breaking dependency chains has when using the random data layout (RND). In that case, major improvements are observed. This indicates that the original version of SPMATVEC is severely affected if the memory access pattern is irregular and that latency caused by the cross-iteration dependency on the pointer traversal is an important factor.

Additional improvements *might* be achieved by other techniques such as struc-

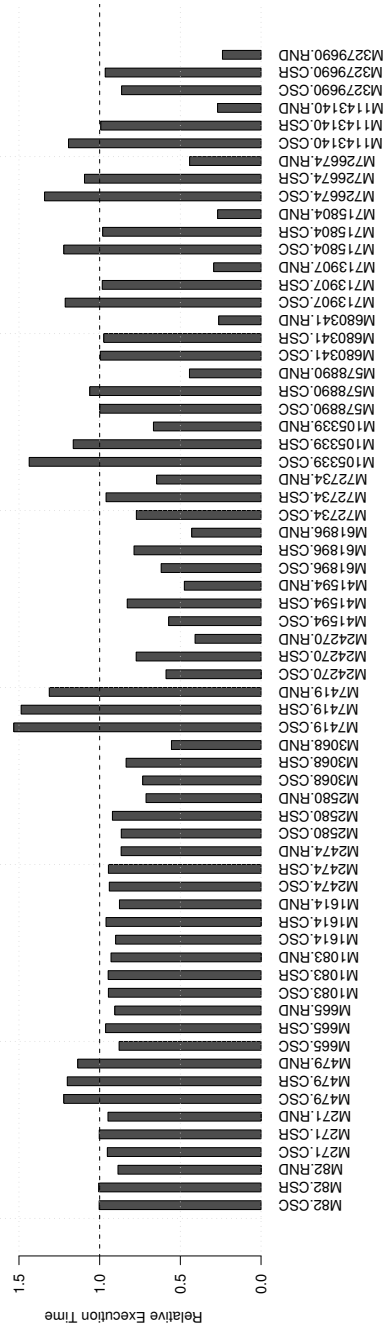



Figure 3.6: SPMATVEC - Relative execution time using prefetched pointer arrays. This figure shows the relative execution time when pointers are not accessed using a pointer chasing loop, but instead are all prefetched into an array. The reference time is the time obtained by running the original pointer chasing loop. In the case that the memory reference stream is regular (CSR is very regular, but CSC is also relatively regular) it is observed that performance is roughly similar. In these cases, the hardware prefetching mechanism is thus able to predict future references of the pointer traversal. With the RND data set this is not so and in this case, the explicit exposure of memory addresses through the pointer array results in significant performance improvements.

| Benchmark | Abs. CPI | Abs. L2 Miss Rate | Abs. Data Bus Utilization |
|---|---|---|---|
| MCF | 1.00-4.79 | 0.00-0.04 | 0.12-0.25 |
| ISPMATVEC | 0.45-1.26 | 0.00-0.02 | 0.00-0.34 |
| IJDSPMATVEC | 0.53-1.56 | 0.00-0.03 | 0.00-0.34 |
| IJACIT | 0.42-1.42 | 0.00-0.02 | 0.00-0.33 |
| ASM | 0.67-2.34 | 0.00-0.03 | 0.00-0.28 |
| TRMAT | 0.40-5.84 | 0.00-0.03 | 0.00-0.27 |
| CMCK | 0.33-1.14 | 0.00-0.00 | 0.00-0.12 |
| MPERM | 0.43-1.98 | 0.00-0.04 | 0.00-0.43 |

Table 3.4: Summary of indirection array based benchmarks and MCF. The table lists absolute CPI (cycles per instruction) absolute L2 cache miss rate and absolute data bus utilization.

| Benchmark | Rel. Execution Time | Rel. L2 Miss Rate | Rel. Data Bus Utilization |
|---|---|---|---|
| SPMATVEC | 0.67-14.96 | 1.51-18.31 | 0.20-3.89 |
| SPMATMAT | 0.79-0.85 | 0.29-3.94 | 0.60-8.38 |
| JACIT | 1.04-14.24 | 2.38-13.58 | 0.28-2.73 |
| DSOLVE | 0.05-0.32 | 3.71-27.74 | 1.63-46.72 |
| PCG | 1.09-4.07 | 5.03-69.12 | 0.46-50.37 |
| ISPMATVEC | 0.50-2.57 | 0.40-2.63 | 0.64-3.28 |
| ISPMATMAT | 0.77-0.84 | 0.08-0.71 | 0.18-1.49 |
| IJACIT | 0.85-2.50 | 0.71-3.36 | 0.59-3.03 |

Table 3.5: Summary of dense vs. sparse benchmarks. The table lists range of relative execution time, relative L2 cache miss rates and relative data bus utilization with respect to the dense implementation.

ture splitting. Curial et al. [35, 36] have shown speedups of 2.07 and 1.72 using structure splitting on the POWER4 and POWER5 architecture, respectively. They implemented structure splitting in the IBM XL compiler. Hagog and Tice [50] have implemented structure splitting in GCC and show performance improvements on the *art* benchmark from SPEC2000 of 1.44 and 1.50 on the G4 and G5 architectures, respectively. In other cases their methods did not show any significant effect. We realize that their experiments were evaluated on different architectures than we used, but it gives an indication of what can roughly be expected from such techniques.

Above, the potential performance improvements have been shown when memory address streams are made predictable, either implicitly by managing the layout of a data structure, or explicitly by exposing the memory addresses through a prefetch array. The potential improvements of managing reference streams have shown to be more influential than techniques like structure splitting. These findings support Thesis 2, that the unpredictability of memory reference streams is largely responsible for performance degradation in pointer-based applications. Therefore, memory reference streams must be made predictable, and this should be a guiding principle when considering compiler optimizations targeting irregular code.

### 3.5.3 Memory Bandwidth in Irregular Applications

In data intensive regular applications, memory bandwidth is the main factor constraining performance. In contemporary processor designs, the memory bus is shared among multiple cores. Performance of the memory system is measured using the *data bus utilization*, which is defined as follows [62]:
"Data bus utilization is the percentage of bus cycles used for transferring data among all bus agents in the system, including processors and memory." Levinthal [77] states that the bandwidth on an Intel Core 2 system is affected by numerous factors and that for each system, the maximum bandwidth should be determined by benchmarking a triad ($A[i] = B[i] + a * C[i]$). For the system used in our experiments, the data bus utilization when executing a triad on a single core is 0.4746.

Figure 3.7 shows the data bus utilization for JACIT. Compared to the upper bound measured, the data bus is not fully utilized. The memory layout CSR scores consistently better than the CSC and RND layouts. This fact is also reflected in the L2 cache miss rates, which are shown in Figure 3.8. In this figure, the miss rate is subdivided into *demand misses* and *prefetch misses*. A *high* demand miss rate indicates that the prefetchers are not able to predict future references and therefore cause a cache miss when they are actually needed. Prefetch misses occur whenever a prefetch request from the L1 cache is received and the data is not found in the L2 cache. Whenever the memory references follow a very regular pattern (e.g. CSR input order), practically no demand misses occur, which is a good thing. CSC input order performs slightly worse than CSR, which is due to the fact that memory accesses do exhibit a good locality, there is still some regularity and demand misses still do not occur frequently. For the RND input order, this is different. Both prefetch and demand misses increase dramatically and this is reflected in the normalized execution times, as shown in Figure 3.9. A summary of the data bus utilization measured for

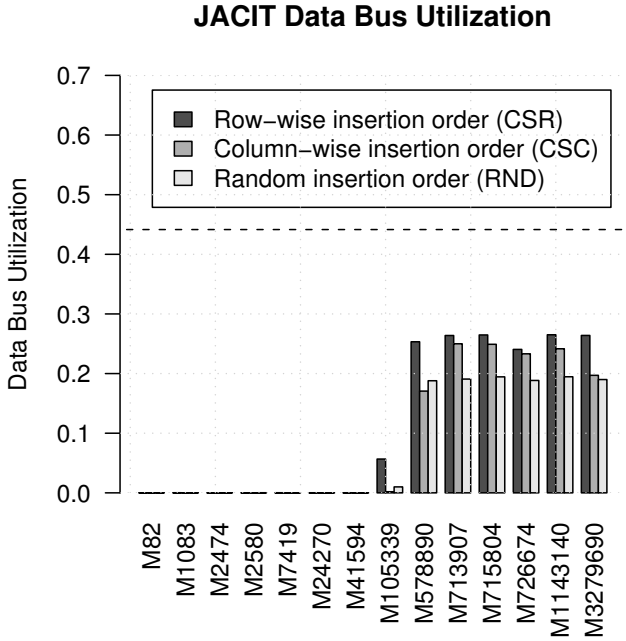



Figure 3.7: JACIT - Data Bus Utilization. On the x-axis are the different data input sets, ordered by increasing size from left to right. As long as the input sets fit in the caches, the data bus is not used. As soon as L2 cache misses start to occur though, the data bus utilization rises quickly. In this case, the data layout has a large influence on performance (See Figure 3.9). Therefore, when the RND set is used, much of the data fetched from main memory is not actually needed.

the benchmarks are shown in Table 3.3, 3.4 and 3.5.

The results show that in nearly all cases, the bandwidth is not fully utilized. As shown in the previous section, breaking dependency chains by explicitly exposing the memory reference stream to the processor can be a rewarding technique. This supports Thesis 3. There is extra bandwidth available and putting this to work to break dependencies potentially increases performance.

This extra available bandwidth should be put to use to resolve dependency information at run-time. Techniques that come to mind first are prefetching techniques in which pointer chains are traversed and stored in arrays prior to the actual traversal [65, 81]. Other techniques that effectively break dependency chains by reordering data [113] also benefit from extra available bandwidth and also fit in this category.

In Section 3.6.2 the bandwidth characteristics when running multiple processes concurrently on the different combinations of cores will be discussed.

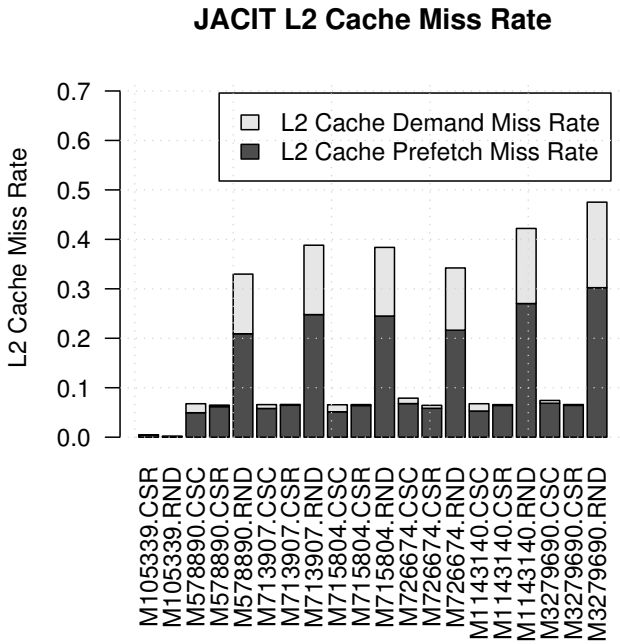



Figure 3.8: JACIT - L2 Cache Miss Rate. On the x-axis are the different data input sets, ordered by increasing size from left to right. Only data sets for which L2 caches do occur are depicted. It is very clear that the CSR data layout performs best, as this data layout does not result in any demand misses. The RND data set shows most L2 cache misses, both prefetch and demand misses. Apparently, the prefetching mechanism is often triggered, but most likely, the data fetched is not actually used and actual data references often result in a demand miss.

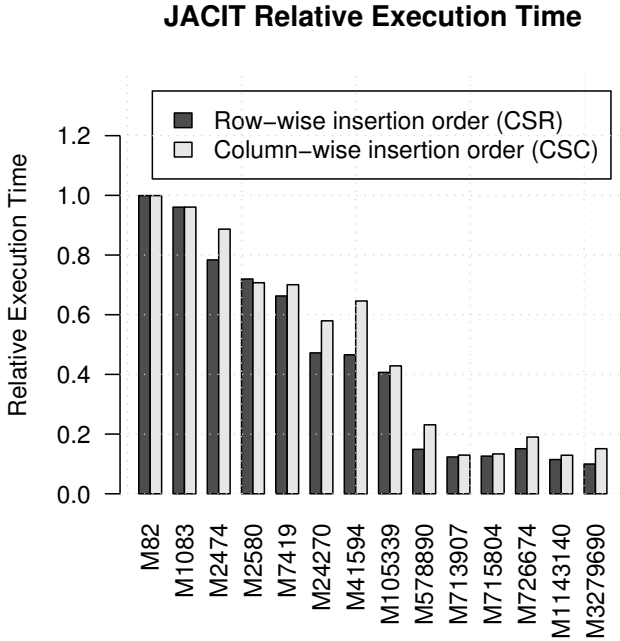



Figure 3.9: JACIT - Relative Execution Time. On the x-axis are the different data input sets, ordered by increasing size from left to right. The reference time is the time obtained by running the benchmark using the RND data sets. At the point where the relative performance of the two regular layouts perform best, the data bus utilization is also high. In other words, as soon as the main memory is accessed, regularity in memory reference streams becomes more important.

### 3.5.4 Controlling the Impact of Irregularity

Thesis 4 states that much of the impact of irregularity can be controlled by choosing an appropriate data layout. The results of the experiments suggest that major performance improvements can be achieved by controlling data layout. For example, the results in Figure 3.5 showed that for larger data sets, restructuring memory to match the traversal order can result in speedups of up to 8.5. These are significant improvements. While DSOLVE did not yield great speedups for its input set (see Figure 3.4), for reasons explained in Section 3.5.1, the results of the other benchmarks suggest that if restructuring would be done after LU factorization, then similar improvements would be achieved for these data sets as well (see Figure 3.5 and Table 3.3). In other cases where restructuring may prove difficult, explicit exposure of memory addresses did yield significant speedups, as was shown in Section 3.5.2. Both approaches deal with the problem of irregularity by changing access to the data, either by explicit restructuring or by providing extra information to expose memory addresses to the processor without being hampered by dependency chains. Other data restructuring approaches have also been shown to improve performance considerably [113]. Therefore, in accordance with Thesis 4 we state that much of the impact of irregularity can be controlled by techniques as mentioned above.

### 3.5.5 Irregularity of Sparse Code

Thesis 5 states that contrary to common belief sparse code is not necessarily very irregular. It will be shown here, by comparing dense kernels with sparse kernels, that their behavior is not necessarily radically different from their dense counterparts. IJACIT is presented in more detail and results for the other comparisons are summarized in Table 3.5.

In certain cases, indirection array based applications can show performance that is near the performance of their dense counterparts. This is because the matrix structure itself is traversed in a regular fashion. This occurs for instance in IJACIT, which executes the following loop structure every iteration:

```
for( i = 1; i <= Size; i++ ) {
    x_2[i] = b[i];
    j1 = ia[i];
    j2 = ia[i+1]-1;
    for( j = j1; j <= j2 && ja[j] < i; j++ ) {
        x_2[i] -= a[j] * x_1[ ja[j] ];
    }
    diag = a[j];
    j++;
    for( ; j <= j2; j++ ) {
        x_2[i] -= a[j] * x_1[ ja[j] ];
    }
    x_2[i] = x_2[i] / diag;
}
```

Only array $x\_1$ is accessed irregularly. If the dimensions of the matrix are not too

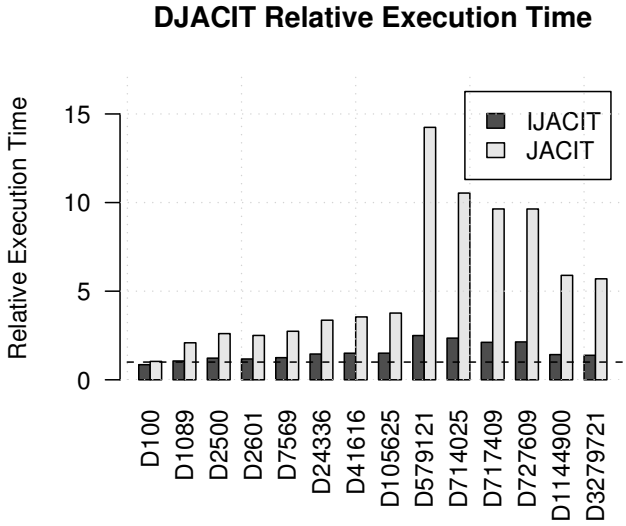



Figure 3.10: Dense vs. Sparse Code - Relative Execution Time. This figure shows the relative execution time of IJACIT and JACIT, with DJACIT used as a reference. These benchmarks are identical (for IJACIT and DJACIT this includes their data layout), except that IJACIT and JACIT contain indirect accesses to the data. Therefore, this figure gives an estimate of the impact of the overhead of these data access methods. For IJACIT, it shows that there is some overhead, but this is not always as large as often believed. For JACIT, the overhead is much larger. Cache performance is a large factor, as is seen for the data set $D579121$ where the cache performance of especially JACIT is relatively bad.

large, this vector will remain in the cache. Also, while the access pattern of the right hand side vector may be completely unpredictable at compile-time, it will often show good locality due to the structure of the matrix. Another characteristic found in indirection array based applications is the presence of loop bounds that are data dependent. This potentially prevents a compiler from doing reordering and parallelizing transformations.

To measure the overhead incurred by indirection arrays, we compare the execution times of dense implementations of some benchmarks against their indirection array based counterparts. The data sets they operate on have the same dimensions, but the indirection array based version adds overhead by introducing an indirection array that stores column indices. This array is used to access another array, in the case of IJACIT this is $x\_1$.

Figure 3.10 shows the relative execution times of DJACIT and IJACIT. It is interesting to see that the dense version does not execute that much faster, especially

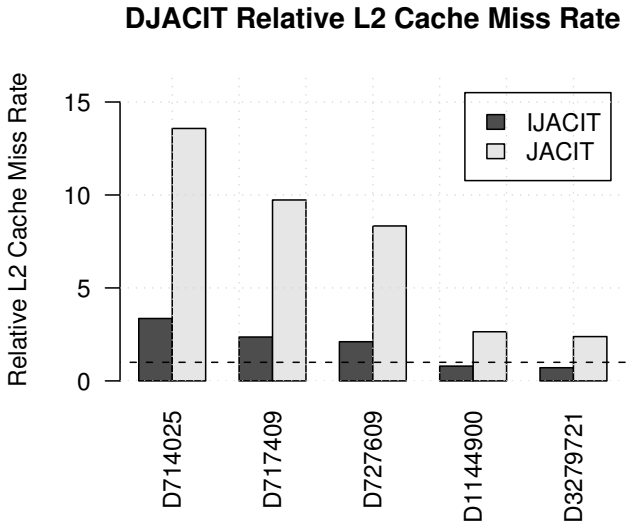



Figure 3.11: Dense vs. Sparse Code - Relative L2 Cache Miss Rate. Only data sets are shown for which in both cases a significant amount of cache misses occur. For the smaller data sets, IJACIT has a higher miss rate that DJACIT, which is due to the smaller working set of DJACIT (which does not use the indirection array). JACIT shows a much higher miss rate and this is due to the fact that the pointer structure nodes contains data that is not used by the algorithm which wastes cache space. Eventually, for the largest data sets, DJACIT has a higher miss rate than IJACIT but it does not have to fetch the index data and therefore still performs better than IJACIT (See Figure 3.10).

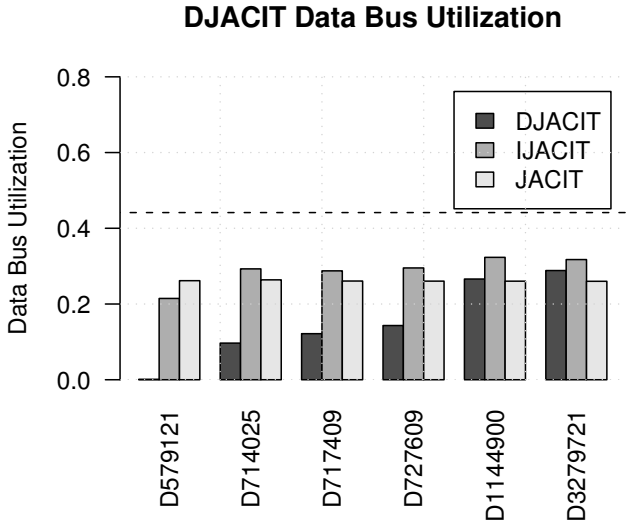



Figure 3.12: Dense vs. Sparse Code - Data Bus Utilization. For the smaller data sets, IJACIT and JACIT show a higher data bus utilization, as there is more data to fetch for these benchmarks (index array and data from pointer nodes). For the larger data sets, DJACIT and JACIT achieve a data bus utilization that is quite close to that of IJACIT but for DJACIT, this does not include the index array. Therefore, DJACIT will execute faster in these cases, which is confirmed in Figure 3.10.

for the smallest and largest data sets. Note that neither of these implementations have been hand-optimized and thus the compiler is responsible for the generation of efficient code. For DJACIT, the Intel C compiler generates vectorized, unrolled and multi version code. For IJACIT, however, none of these optimizations are performed. This is due to potential data dependencies (dynamic loop bounds) and non-standard loop headers. Even if dependencies are ignored [2], the optimizations are still not performed, due to the indirection in the inner loop[3]. Despite the lack of advanced optimizations, IJACIT does not perform dramatically worse than DJACIT.

For the medium sized data sets, however, the dense version is considerably faster. There is a peak in performance difference between DJACIT and IJACIT, which is largely due to the fact that IJACIT also uses an indirection array that is used as an offset in the array $x\_1$. This indirection results in extra data traffic. The point where the execution times show the greatest difference is the region in which IJACIT starts to suffer from L2 cache misses, whereas for DJACIT the data set still fits in L2 cache. If we look at the relative L2 cache miss rate in Figure 3.11, this becomes clearer. Results are only shown for matrices that have L2 cache misses for both versions. For matrix *D761*, IJACIT suffers from L2 misses, whereas DJACIT has almost no cache misses (and therefore is not shown in the graph). As the data sets grow larger, DJACIT starts to suffer from misses as well, and the performance differences become much less. The decomposition of L2 cache misses in prefetch and demand misses shows that for both benchmarks, demand misses do not occur. Thus, prefetched data arrives in time in the L2 cache. Data bus utilization is similar for both benchmarks when using the largest data sets. But in the case of IJACIT, part of the data traffic is caused by the indirection array, which is not part of DJACIT. To conclude, DJACIT runs faster because of less data traffic, no irregular access to the right hand side vector and the fact that optimizations can be applied.

For the pointer-based version (JACIT) the performance figures are much worse, also in the case that the working set of the dense version is also too large to fit in the cache. While data bus utilization is similar, the effectiveness of pointer-based applications is less, as much of the data brought into the cache remains unused (only a few members of a structure node are actually used).

The lesson that can be learned from these experiments is that irregularity is not always a very constraining factor. The example of IJACIT showed that performance degradation is not as severe as often is believed, supporting Thesis 5. The comparison of the other algorithms is shown in Table 3.5 and they show similar behavior. Pointer-based applications make this problem a bit harder. In that case, regular behavior can be forced by choosing the proper layout. Together with recent techniques implemented in compilers, such as *structure splitting*, even pointer-based applications might in certain cases reach performance levels previously only believed to be possible for dense applications.

[2] Using *#pragma ivdep*

[3] This information can be obtained by using the *-opt-report* option.

### 3.5.6 Optimizing Compilers

The SPARK00 benchmarks contain relatively simple kernels. However, even the most basic constructs that are responsible for irregular behavior prevent the application of many compiler optimizations. Thesis 6 states that traditional code and control flow optimizations do barely have a positive effect on irregular code. In the experiments in this section, we will show the effectiveness of the standard optimization levels of the Intel C Compiler (O1, O2 and O3) on both regular and irregular code.

Timings have been obtained for all benchmarks of SPARK00 for the options mentioned above. For each benchmark that uses a matrix as input data set, two different matrices are used, namely *M24270* and *M715804*. They will be referred to as the small and large input set, respectively. For DSOLVE, *M24270* and *M105339* are used. In DSOLVE, a LU-factorized matrix is used, and *M105339* has a similar size as *M715804* after LU-factorization.

Figure 3.13 shows the speedups obtained by the optimization levels -O2 and -O3, relative to the execution time using the setting -O1. It shows that for irregular applications, there is not much to be gained by the extra optimizations that -O2 and -O3 provide. The dense codes do show some additional performance increase when -O2 and -O3 are applied.

These results indicate that irregular code itself is hard to optimize, if only code and control flow optimizations are considered. In the experiments just described, the data layout was fixed, and only the quality of the target assembly code was responsible for any performance improvement (or degradation). As was shown in Figure 3.5, if the code is not changed but the data layout can be changed, then the potential performance improvements are much higher. Breaking dependency chains (either by prefetching of pointers or by data layout remapping) also showed large potential performance improvements (Figure 3.6). It can be concluded that speedups obtained by such methods are much greater, and therefore attention should shift towards data restructuring transformations instead of code and control flow optimizations, as stated in Thesis 6.

## 3.6 Experiments on Multiple Cores

This section describes two sets of experiments. First, we will focus on the effect of irregularity in applications on multi-core platforms. Second, the bandwidth characteristics of such applications are considered. Finally, a short summary of the results is given. All figures contain information on processor efficiency, measured by the (relative or absolute) clocks per instruction retired (CPI), and data bus utilization, which is the fraction of bus cycles used to transfer data. Each run in separated by a dotted line. Within each run, each different process is depicted using one black data point, which shows the bus utilization, and an associated grey bar, which shows the CPI. Whenever applicable, the legend shows which data set is associated with a data point. The grey data points show the estimated aggregate data bus utilization, in case cores from two different chips are used simultaneously. This is explained in more detail in Section 3.6.2. Each run is annotated with the configuration number.

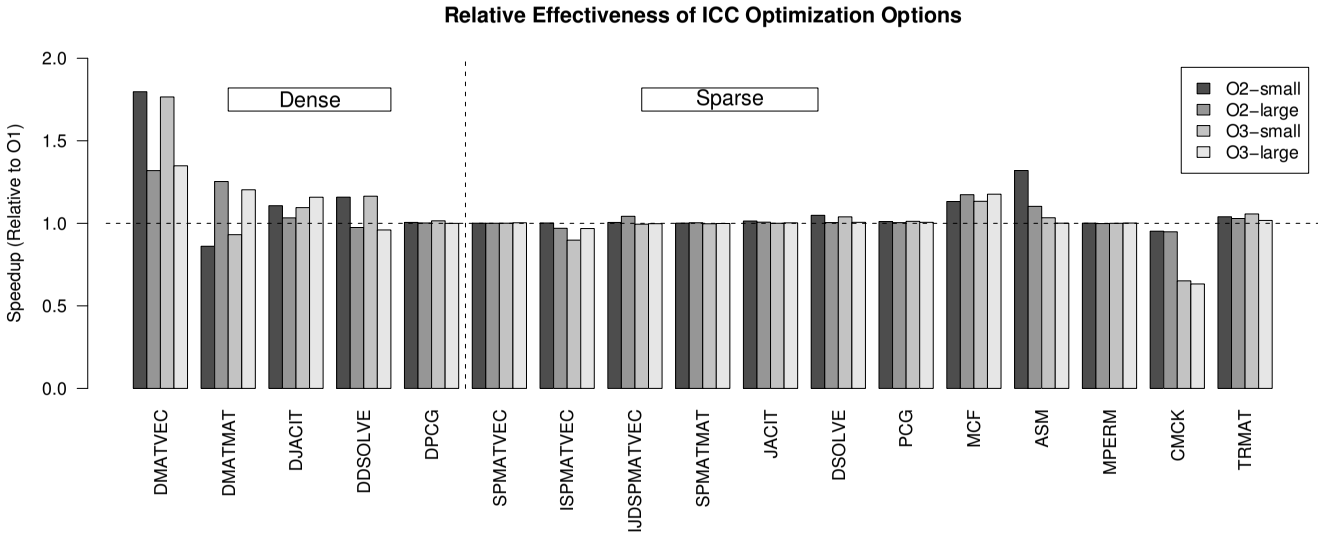



Figure 3.13: The effect of standard compiler options of the Intel C compiler. The benchmarks are all run using a small and a large data set. For the benchmarks using matrices as their input set, $M24270$ and $M715804$ are used. DSOLVE uses $M105339$ as its large data set. The dense benchmarks use $D24336$ and $D717409$ as its data sets. DDSOLVE (dense version of DSOLVE) uses $D88209$ and $D707281$. The speedups given are relative to the execution times observed using -O1. For the dense benchmarks, the optimization levels -O2 and -O3 do in some cases give additional performance. The irregular benchmarks do not really benefit from these optimization levels in general, except for MCF and ASM. However, the speedups as observed for DMATVEC are not observed in any of the irregular benchmarks.

Descriptions for the configurations can be found in Table 3.2. Note that as described in Section 3.3.2 our experiments are run using two different memory access patterns. If different memory access patterns are used, the column is annotated with either an 'S' or an 'R', indicating the use of the sequential or random reference patterns, respectively.

### 3.6.1 Irregularity on Multi-Core Systems

On a single-core configuration, irregularity in memory reference streams has little impact, if the working set size of the application does not exceed the cache sizes [117]. In order to verify this in the case of multi-core configurations, both data sets that fit in the cache and sets that do not fit in the cache are used. The evaluation of different memory access streams is realized by running the same program with the two different memory layouts, sequential and random. This is only possible for the benchmarks which use pointer structures to represent the matrices. By controlling the order in which elements are added to the matrix, different memory access patterns emerge when traversing the pointer structures, showing different behavior.

When running PCG with a dataset that fits into L1 cache (*impcol_b*), no performance degradation is observed when running on different combinations of cores simultaneously. Data layout does not affect performance in this case. The same holds for sets that fit into L2 cache (*bcsstm34* and *cage9*). In contrast, for data sets that do not fit in the cache (*G2_circuit* and *nd3k*), performance differences are observed among different core combinations and memory layouts. This is shown in Figure 3.14. Thus, whenever possible, cores should be selected from different chips and if cores from the same chip are to be used simultaneously, choosing the *fast* combinations (as indicated in Table 3.2) has a significant impact on performance.

While the absolute performance numbers of PCG are much better in the case that memory access is chosen to be regular (the runs marked with 'S'), the relative performance impact when running multiple instances is much greater compared to the impact when memory access is random (the runs marked with 'R'). Regular behavior of an application is a desirable property, but not all data structures can be designed to show this behavior (e.g. unpredictable graph traversals). In cases where data access cannot be optimized, the single-core performance might be disappointing, but such applications will scale relatively better than regular applications when running multiple instances concurrently.

Therefore, it can be concluded that as long as data sets fit within the caches, performance does not suffer when running multiple instances of an irregular application simultaneously, but when the data does not fit in the caches, the choice of a combination of appropriate cores has a significant impact.

### 3.6.2 Memory Bandwidth on Multi-Core Systems

In data intensive regular applications, memory bandwidth is the main factor constraining performance. In contemporary processor designs, the memory bus is shared among multiple cores. While it has been shown in Section 3.5.3 that most irregular

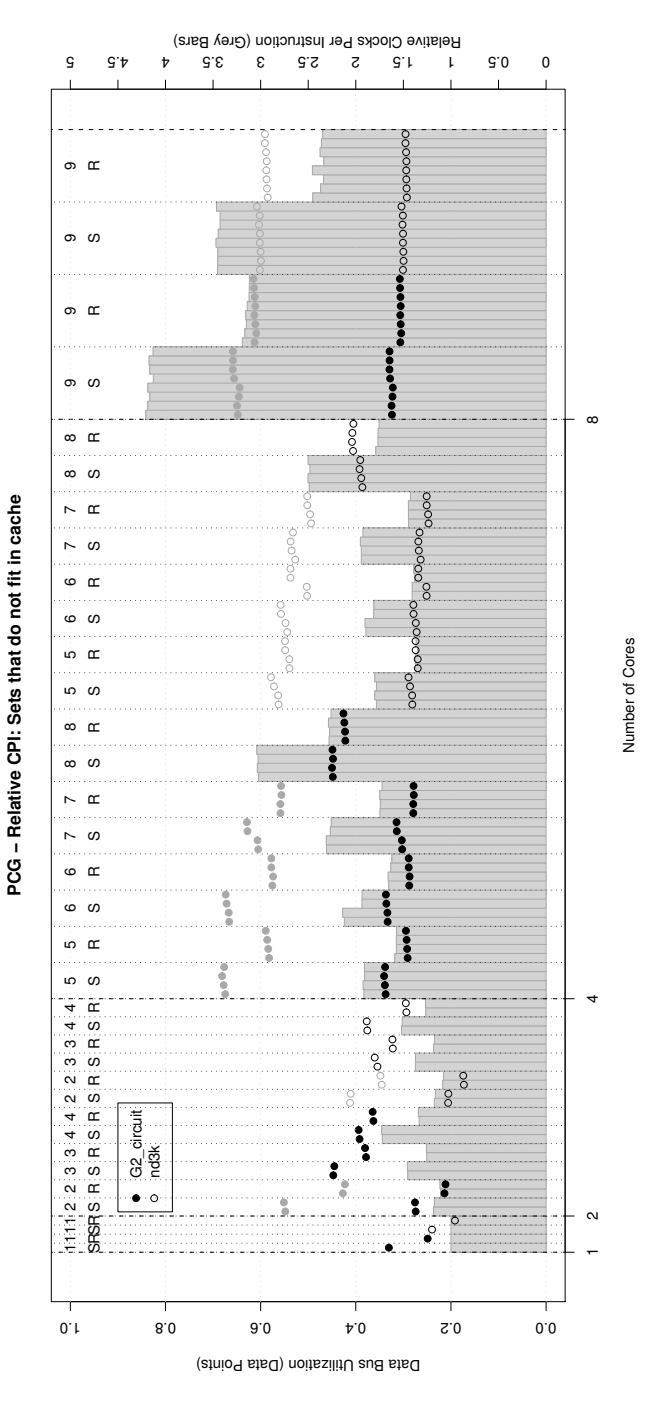



Figure 3.14: PCG - Performance in relative clocks per instruction retired (CPI). Depicted are the two sets that do not fit in L2 cache when run on a single core *(G2_circuit, nd3k)*. The bars show the relative CPI. The data points show the data bus utilization. The description for the core configuration numbers shown in each column can be found in Table 3.2. It is clear that starting at 4 cores, the bandwidth is already fully utilized and adding more processors with high bandwidth requirements will not increase the aggregate throughput.

applications do not fully utilize the bandwidth in single-core configurations, we will show that for multi-core configurations, not only memory latency is a problem, but also memory bandwidth. Performance of the memory system is measured using *data bus utilization*. We repeat the definition given above [62]:
"Data bus utilization is the percentage of bus cycles used for transferring data among all bus agents in the system, including processors and memory." For the system used in our experiments, the maximum bandwidth for multi-core configurations is done by benchmarking a triad ($A[i] = B[i] + a * C[i]$) for all core configuration mentioned in Table 3.2. The results are shown in Figure 3.15. The black data points indicate the data bus utilization as reported by Intel PTU, per core. The data bus utilization is reported per chip, and therefore, for instance for configuration 2 (2 cores, different chip), it seems that the bus utilization is lower than for configuration 3 and 4, while the measured performance in CPI is better. In fact, the data bus utilization is also higher, but the cores only report the data bus traffic for the chip they are part of. Therefore, as the 2 chips and the application that runs on them are identical, we estimate the actual aggregate bandwidth by multiplying the measured bandwidth by 2. These estimates are depicted by the grey data points.

Maximum bandwidth utilization can be viewed per chip, or for the entire system. For a single chip, the maximum bandwidth measured is 0.4746. While measurements on a triad should give an upper bound of performance [77], we have observed higher values using MPERM, namely 0.5029. For a single chip, the maximum bandwidth available seems to be roughly 50%. If we look at the highest measured value for data bus utilization in the case that 2 chips are used and multiply this by 2, we reach a maximum aggregate data bus utilization of 0.7076. For MPERM, 0.7571 is the estimated maximum data bus utilization. Therefore, the maximum bandwidth as obtained by MPERM is used as a reference.

Figure 3.16 shows the data bus utilization for MPERM. The grey bars show the absolute CPI and the data points show the data bus utilization. The benchmark MPERM achieves quite a high data bus utilization when running on a single core. As expected, the small data sets that fit in the caches (*LF10* and *bcsstm34*) do not cause traffic on the memory bus and therefore their CPI per process does not show any performance degradation. For the large data sets (*G2_circuit* and *nd3k*), the performance is affected when moving to multi-core configurations. Near maximum bandwidth utilization is already reached in some cases when running on 2 cores and scaling further significantly slows down performance, as indicated by the increasing CPIs.

Figure 3.17 shows the data bus utilization for JACIT. MPERM only uses the sequential input data sets, whereas JACIT also uses the random data sets. The irregular memory reference pattern caused by the random data set does not give large differences in data bus utilization, especially when moving beyond the single-core configuration. There is a large difference in performance though, as indicated by the measured CPI (scale is different from the scale in Figure 3.16). While the amount of data traffic is similar for both the regular and irregular memory reference streams, much of the data fetched is actually not used in computations when the access pattern is irregular.

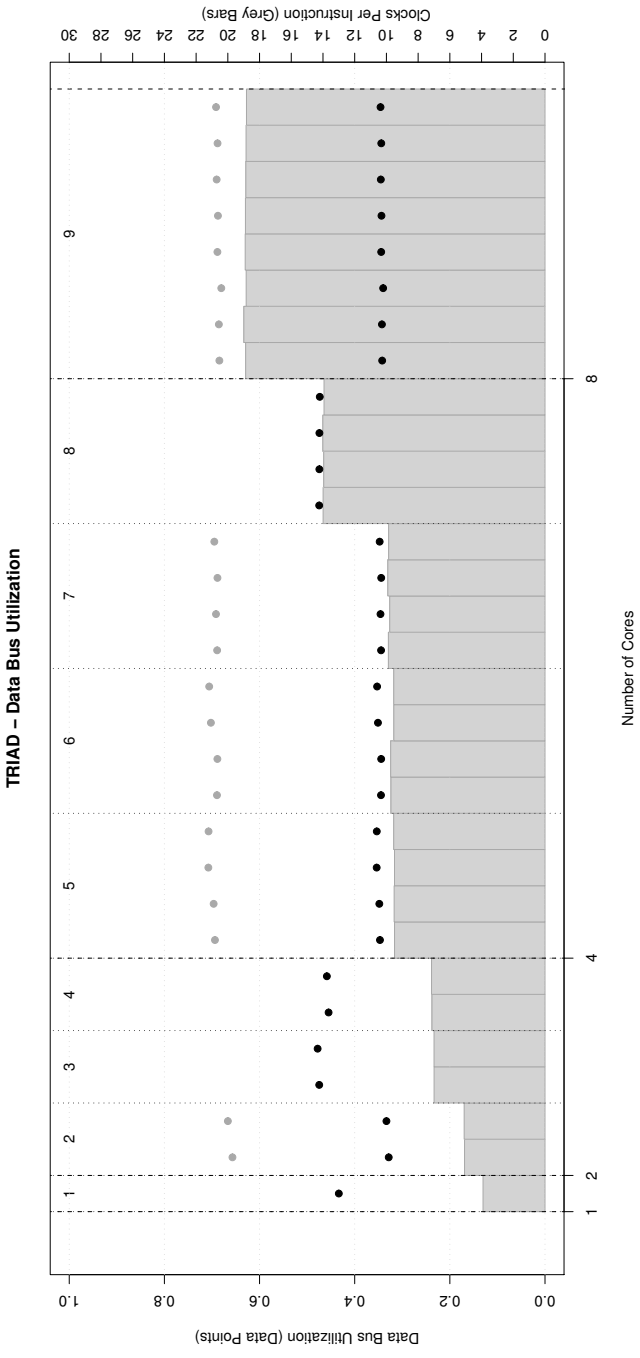



Figure 3.15: TRIAD - Data bus utilization. The bars indicate the absolute clocks per instruction retired, the black data points data bus utilization. The grey data points show the estimated aggregate data bus utilization when cores from distinct chips have been selected.

Figure 3.16: MPERM - Data bus utilization. The bars indicate the absolute clocks per instruction retired, the black data points data bus utilization. The grey data points show the estimated aggregate data bus utilization when cores from distinct chips have been selected. Data for all input sets is shown.

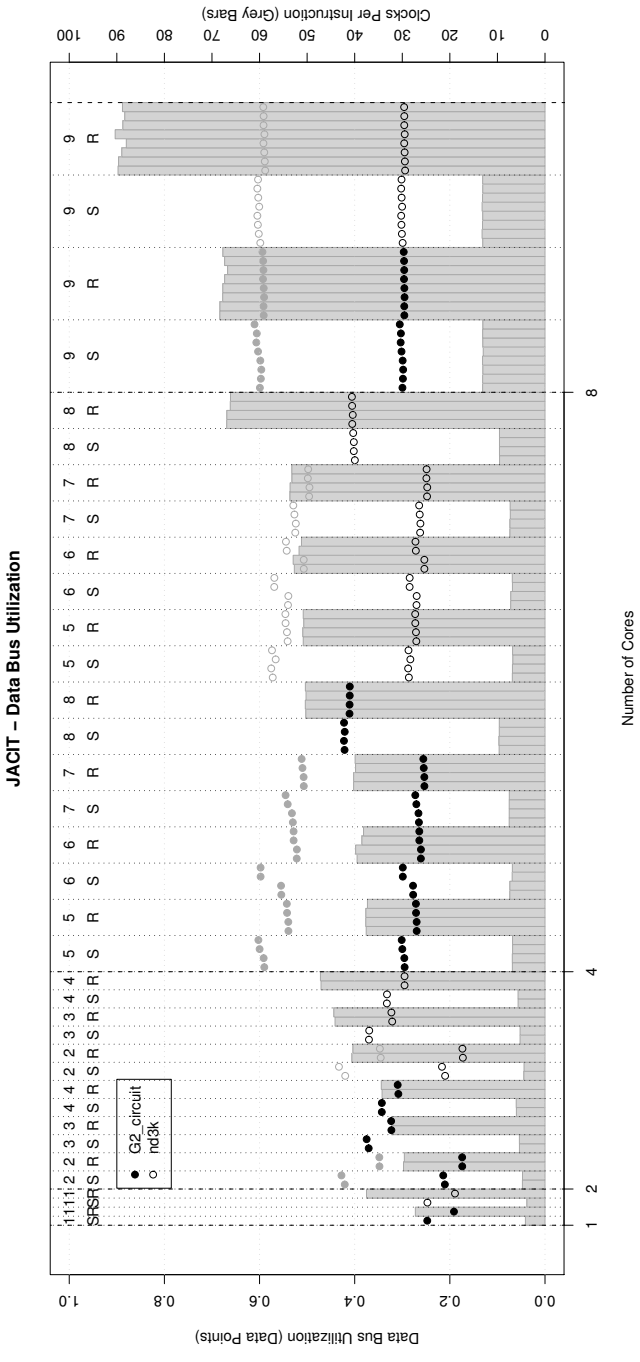



Figure 3.17: JACIT - Data bus utilization. The bars indicate the absolute clocks per instruction retired, the black data points data bus utilization. The grey data points show the estimated aggregate data bus utilization when cores from distinct chips have been selected. Only data for data sets that do not fit in the caches are shown.

## 3.7 Summary

The performance degradation caused by different sources of irregularity has been characterized using the SPARK00 benchmarks. The study has been performed on the widely used Intel Core 2 architecture. Guided by the theses put forward in Section 3.2 we have described the results of our experiments in Section 3.5 and 3.6. We summarize our findings here.

Different aspects of irregularity have been evaluated and we have been able to measure the impact of irregularity in memory reference streams for both pointer-based and indirection array-based applications. We can conclude that as expected, pointer-based applications are not in general suffering because of the pointers, but that it is the irregularity of the memory streams that is the problem for the performance of these programs.

Irregularity does not necessarily have a large impact on performance. Our experiments have shown that for smaller working set sizes, the degree of irregularity in memory reference streams did not have any significant effect (Thesis 1). This is independent of the number of cores that are used. For larger sets, the influence of irregularity becomes more dominant and the layout of the data does make a difference. This was observed across all pointer-based benchmarks. In this case, the use of multiple cores will incur a degradation in performance and also the choice of which cores are used does matter.

In the case that irregularity does have an influence, a major factor contributing to the behavior of an application on a specific architecture is the predictability of the memory reference stream (Thesis 2). In pointer-based applications, latency caused by dependent memory references has a significant impact. It was shown that if the hardware prefetching mechanism fails (random data references), explicitly exposing the memory addresses by using pointer prefetch arrays did result in significant performance improvements.

As memory latency often occurs in irregular applications, the full potential of the memory subsystem is not used. This was reflected in our experiments, which showed that in nearly all cases, the data bus is not fully utilized when running irregular applications. Given the results described above, we propose that this extra available bandwidth should be exploited to communicate dependency information (Thesis 3). For multi-process configuration, this is not necessarily the case. While latency is a problem per process, the processes together easily consume all available bandwidth (Thesis 7).

The first three theses focused on quantifying the behavior of irregular applications. The results that have been obtained indicate that the performance impact of irregularity can vary considerably. It was shown that by controlling data layout, many of the performance bottlenecks can be solved (Thesis 4).

Thesis 5 addressed the supposed irregularity of many sparse algorithms. We have shown that the impact of this irregularity is not as severe as is often believed. Performance of sparse algorithms using indirect addressing has been compared to their dense counterparts and performance differences have been found to be relatively small and are mostly explained by increased working set size due to the extra index array.

All results described above show that for irregular applications, data layout is a very important factor. In addition, we have shown that optimizations that target code optimization do not yield any additional improvement for irregular code (Thesis 6). Therefore, we expect that the traditional approaches are not very likely to cause a breakthrough in the optimization of irregular code. On the other hand, data restructuring approaches can result in significant performance improvements and hence focus in compiler research should shift towards data restructuring approaches.

Thesis 7 addressed the main performance issue arising when irregular memory intensive applications are run concurrently: while irregular applications underutilize the memory subsystem in a single-core configuration, this is not the case when moving to multi-core platforms. Regardless of the fact whether the memory reference streams are regular or irregular, the memory bandwidth is easily saturated.

While for small data sets, the multi-core platform performs extremely well this is not true for data intensive irregular applications. Many factors have an influence, such as the selection of cores to run on and the data layout used. The evaluation performed in this chapter only considers running multiple independent processes. Even in this case, optimal performance cannot be obtained. Distributed irregular applications where dependencies between processes exist will make this problem even harder to tackle. Therefore, both computer architecture and compiler technology should be focusing on the dynamic optimization of data layout to handle the complex interactions of irregular applications on multi-core platforms.

The work presented here defines performance bounds of what can be achieved by restructuring transformations. Ideally, no matter what data layout is provided initially, the application generated by the compiler should show the same performance. This is what compilers targeting irregular code should aim for. As of today, such restructuring approaches have not found their way to mainstream compilers yet, but some experimental implementations of transformations targeting data layout have been implemented [35, 48, 50, 68, 73, 75, 79], which is a promising step in the right direction.

# CHAPTER 4

## Concepts of Restructuring Pointer-Linked Data Structures

Irregular memory accesses, especially those caused by the use of pointer-linked data structures, can have a great impact on performance, as shown by the SPARK00 benchmarks in the previous chapter. Therefore, transparent reordering of data structures has great potential. In this chapter, the concepts underlying the restructuring of pointer-linked codes are explained. In subsequent chapters, an implementation is described which employs the LLVM framework, whereby the analyses and transformations operate on the intermediate format of LLVM (LLVM bitcode). In this chapter, the focus is on the concepts of data restructuring methods for pointer-linked structures. For clarity, all examples are given in C.

The impact of irregular access is mainly found in two forms: either by using indirect addressing of an array or by using pointers (the impact of both types has been assessed using the SPARK00 benchmarks). Approaches to restructure irregular code to regular code exist [125], but these only work on indirectly accessed Fortran arrays (i.e., an array can be directly addressed as in $A(i)$, or indirectly addressed by using an index array $B$ as in $A(B(i))$).

In languages such as C, irregular access is typically due to the use of pointers, which makes such code very difficult to optimize. Pointers do not only allow irregular access but can also point to arbitrary locations. Also, multiple pointers can point to the same location (aliasing). Furthermore, pointers can be arbitrarily manipulated and allow more complicated structures to be defined, such as linked lists or trees, all of which present considerable optimization challenges. Hind [55] provides a concise overview of pointer analysis techniques.

One very common pointer structure is the linked list. Linked lists represent a sequence of elements where each element is in a completely unrelated memory location, making it difficult for the compiler to optimize the memory access patterns. The

```
while( node != end ) {
  ... = node->Value * B[idx_expr];
  node = node->Next;
}
```

Figure 4.1: Structure of a loop using a linked list and array $B$.

presence of the code that accesses the linked list impedes analysis of the code (for example, loop-carried data dependency analysis) and thus prevents the application of optimizing transformations such as loop interchange or even more drastic code restructuring. This irregular access pattern also poses a problem for the CPU cache, which cannot exploit the locality of subsequent memory references. Computations performed on subsequent items cannot be vectorized, if a linked list is used. This fact is recognized in [11], which describes many of the key points that make C code so hard to vectorize and parallelize.

In this chapter, the concepts of restructuring pointer-linked structures will be explained using C code examples that traverse linked lists. Restructuring of these lists is accomplished by transforming them into an alternative form that is more suitable for further optimization. The concepts outlined have been implemented in a proof-of-concept C to C restructuring compiler and the results of their application to SPMATMAT and PCG benchmarks from SPARK00 are presented, which shows considerable speedups.

## 4.1 Annihilation and Sublimation

In this chapter, we only deal with loop structures that iterate over linked lists. These loops may also use regular arrays which may be indexed by an expression that depends on the linked list. Figure 4.1 shows the structure of such a loop.

These types of loops prevent optimizations such as vectorization. We will show that these loops can be transformed to loops having only array based regular and irregular access patterns. In this form, optimizations are still not possible. This is not a direct consequence of the use of indirect access patterns, but merely a consequence of mixing an indirect access pattern with a direct access pattern or *another* indirect access pattern. Let us illustrate this with an example:

```
for( i = 0; i < m; i++ ) {
    ...
    for( j = 0; j < n; j++ )
        X[i] = X[i] + A[j] * B[C[j]];
}
```

The access pattern of $B$ prevents vectorization of this code. At this point, two different approaches can be taken. Either impose the access pattern of $A$ on $B$, or impose the access pattern of $B$ on $A$. In the first case, $B$ is restructured to follow the

access pattern of $A$, which is induced by $i$. The loop can now be rewritten to use the restructured array $B'$:

```
for( i = 0; i < m; i++ ) {
    ...
    for( k = 0; k < n; k++ )
        B'[k] = B[C[k]];
    for( j = 0; j < n; j++ )
        X[i] = X[i] +  A[j] * B'[j];
}
```

The other possibility is to restructure $A$ such that it follows the access pattern of $B$. This results in the following loop structure, where $A'$ is the restructured array:

```
for( i = 0; i < m; i++ ) {
    ...
    for( k = 0; k < n; k++ )
        A'[C[k]] = A[k];
    for( j = 0; j < n; j++ )
        X[i] = X[i] + A'[C[j]] * B[C[j]];
}
```

For this restructuring to be valid, the access pattern induced by $C[k]$ must be injective. The access patterns of $A'$ and $B$ are the same, but they still are irregular. This issue will be dealt with in Section 4.2.6. The first method is called *annihilation*, the second *sublimation*.

**Definition 1** *If an access pattern $I$ is provably sequential at compile-time and this pattern is enforced onto other arrays, then this is called annihilation.*

**Definition 2** *If an access pattern $I$ is irregular (the array bounds induced by $I$ cannot be determined at compile-time) and this pattern is enforced onto other arrays, then this is called sublimation.*

At a first glance, it is not obvious that this distinction is effective. However, we will show that both methods lead to significant performance improvements. The two methods are complementary, as annihilation uses a pattern whose bounds are known (symbolically) at compile-time, whereas for sublimation the optimization decisions are taken at run-time. Therefore, applying annihilation does not require much run-time overhead, whereas sublimation does require substantial run-time overhead. This does not render sublimation a useless technique, as in some cases data dependencies can make the application of annihilation inefficient (see Section 4.2.8).

Figure 4.2 shows the architecture of our conceptual restructuring framework. The system consists of a compile-time part and a run-time part. At compile-time, code is generated which results into perfectly nested loop structures. In the case of sublimation, these loop structures can be further optimized by the run-time system, using memory access pattern restructuring, the process in which different access patterns are coerced into a common regularized pattern.

This chapter is organized as follows. First, related work in this area is discussed. Next, the transformation steps a compiler must implement are explained, The most important contribution being the description of an access pattern restructuring framework. The other transformations are merely tools to transform code into a code that is suitable for access pattern restructuring. In the subsequent chapters, it will be shown that some of this code and data transformations can also be replaced by compile-time data layout transformations techniques and a run-time based restructuring environment. The different steps are illustrated by their application on a code example which performs matrix multiplication. This algorithm has been transformed by a prototype C source-to-source restructuring compiler (called MTC) and the results of experiments conducted are discussed.

The approach as described here is essentially different from other approaches discussed in Chapter 2. By linearizing access to a linked list, linked list iteration statements are transformed into simple *for* loops of which the access patterns can be restructured by applying techniques similar to those described by Zhao and Wijshoff [125]. This leads to an intermediate code that is amenable to further optimizations that take the characteristics of the underlying data structures into account. taking the data it operates on into account. In the case of sublimation, a partial execution is required followed by a run-time recompilation of part of the code, resulting in highly efficient code for a particular instance of a linked list.

## 4.2 Transformation Steps

This section describes the steps a compiler should take to transform code using a linked list access pattern into code that is optimized for a specific instance of a linked list. An outline of these steps can be found in Figure 4.2. First, we start with preprocessing steps, such that the code is transformed into a normal-form. Using this normalized code, we will show how a dense intermediate is obtained by step-wise rewriting of the code.

### 4.2.1 Normalization

Programming languages often contain complex expressions. In order to make transformations easier, a normalization step is performed before all other steps. In this chapter, all transformations will be done on C code. In case of C, the normalization step involves the conversion of *for* loops to *while* loops, expression flattening, and common subexpression elimination. These steps lead to a form of C code that is easier to transform.

Loops are transformed to *while* loops using the rule

```
for( init; cond; iter ) { block; } ->
     { init; while( cond ) { block; iter; } }
```

The only point of exit of the loop should be controlled by the condition. Control flow statements, such as *continue*, *break* and *return* are not allowed and such loops will

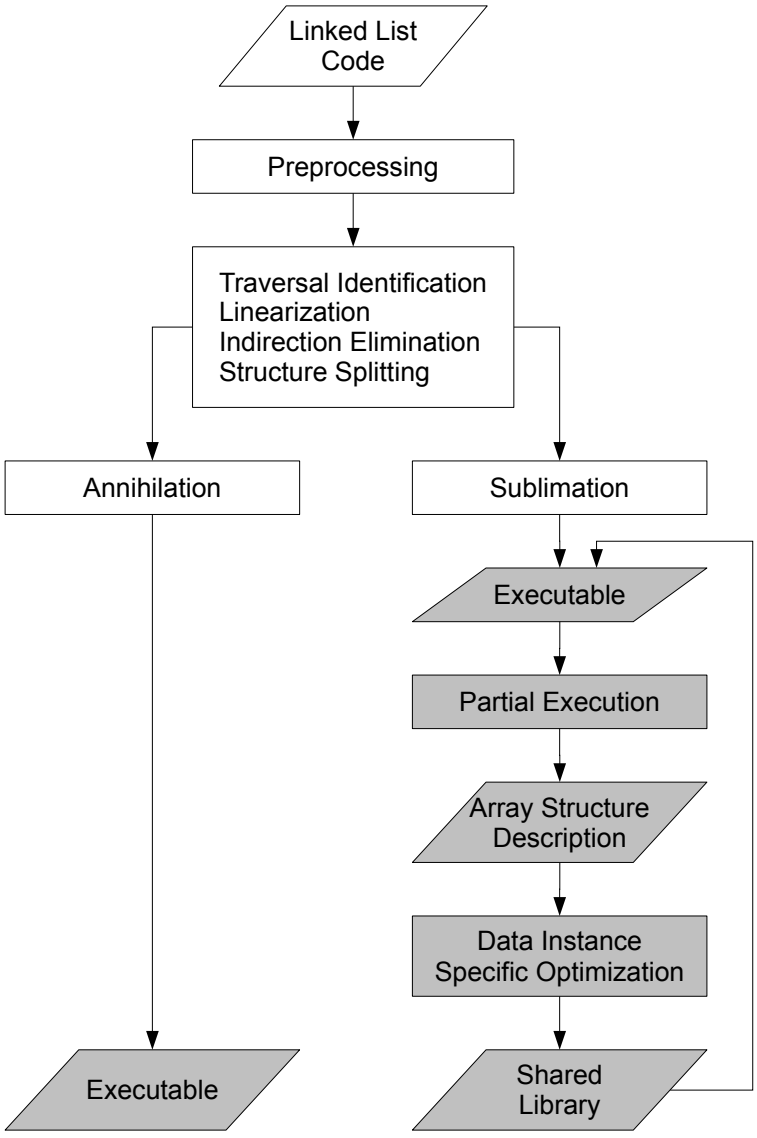



Figure 4.2: Linked list restructuring compiler architecture. The white shapes represent compile-time components and the grey shapes represent run-time components.

not be considered for optimization.

Next, all code is *flattened*. In this form, complicated expressions are unravelled and their results are stored in temporary variables. Also pointer expressions that are dereferenced are first put into a temporary variable. This process can be viewed as creating a type of three address code for C. It drastically reduces the number of cases to consider and enables transformations on, for instance, an array of linked lists.

The last step in the normalization phase is common subexpression elimination which together with the flattening of expressions results in easy identifiable linked list traversals. An example where this transformation enables simple recognition of a linked list traversal is the following code:

```
/* list[x] = list[x]->Next; */
*(list + x) = (**(list + x)).Next;
```

Flattening all expressions results in:

```
temp1 = lists + x;
temp2 = lists + x;
*temp1 = (*(*temp2)).Next;
```

After common subexpression elimination, the result is a code code in which linked list traversals have a simple representation:

```
temp1 = lists + x;
*temp1 = (*(*temp1)).Next;
```

(Note that in the examples shown here non-normalized code is used to keep the examples more readable.)

### 4.2.2 Identification of Linked List Traversals

In order to linearize access to a linked list, linked list access patterns must be identified and formally described. Access path matrices (APM) are matrices that describe every possible traversal path at a particular point in a program [58]. Using these access path matrices, a regular expression that represents the possible traversal paths over a particular code segment can be constructed. A small example is shown here to illustrate how access paths are determined:

```
    void traverse( Node *root, int k )
    {
        int x; Node *p;
        p = root;
L1:
        while( p && p->key != k ) {
            /* Do something */
            ...
            p = p->next;
L2:
        }
        x = p->value;
        p = p->next;
L3:
        while( p ) {
            /* Do something */
            ...
            p = p->next;
        }
L4:
    }
```

At $L1$, the following APM is computed:

| | root | p |
|---|---|---|
| _hp | $\epsilon$ | $\epsilon$ |

The initial definition of $p$ is denoted by $\_hp$ (handle), which denotes a pointer relative to which an access path is computed [58]. At program point $L1$, *root*, $p$ and $\_hp$ refer to the same node.

At $L2$, the list is traversed using the *next* member. As no information about the value of $p \rightarrow key$ is known at compile-time, the assumption is made that the containing while loop executes 0 or more times. This leads to the following APM:

| | root | p |
|---|---|---|
| _hp | $\epsilon$ | $(next^*)$ |

At $L3$, a single pointer has been traversed.

| | root | p |
|---|---|---|
| _hp | $\epsilon$ | $(next^*)(next) = next^+$ |

Finally, the remainder of the pointer list is traversed until the end is reached. We will denote a traversal which terminates at the $NULL$ pointer as $next^>$. At $L4$, the $APM$ looks as follows:

| | root | p |
|---|---|---|
| _hp | $\epsilon$ | $(next^*)(next)(next^>) = next^>$ |

Using these APMs, all access paths can be determined which are used to generate code to linearize these pointer traversals. Proper construction of APMs requires that control flow within the code region of the traversal is regular. Therefore, control flow statements, such as *continue*, *break*, *return* and *goto* are not allowed. Additionally, if side-effects of function calls cannot be determined at compile-time (such as library functions), the traversal will not be considered for optimization.

### 4.2.3 Linearization

Linearization is the process of traversing a linked list and storing the pointers that are encountered during this traversal. The original iteration loop can then be replaced with a *for* loop iterating over the newly created array. All linked list pointers are then replaced with an array reference. Applying linearization on the loop structure from Figure 4.1 results in the following code (memory allocation/deallocation code is omitted):

```
i = 0;
/* Linearize linked list */
while( node != end ) {
  A[i] = node;
  node = node->Next;
  i++;
}
iMax = i;
/* Substitute linearized array for
   iteration pointer */
for( i = 0; i < iMax; i++ ) {
  ... = A[i]->Value * B[idx_expr];
}
```

The loop in which the list is linearized is called the *pre-initialization loop*.

While in this example the linearization is shown as C-code just before the computation loop, this is not necessarily the case in an implementation of a restructuring framework. A method following the concept outlined here has been implemented in a prototype C-to-C restructuring compiler framework (called MTC), but the same effect can be obtained by using a combination of using split memory pools and a restructuring run-time for pointer-linked structures. These techniques and their implementation will be explained in detail in the next chapters.

### 4.2.4 Indirection Elimination

The linearization step produces an array with pointers to a linked list element. Using these pointers, data members are accessed and used in the computation loop. This indirection can be removed by changing the pre-initialization loop and performing the indirection there. Thus, the pre-initialization loop will acquire an extra level of indirection, but in the computation loop, one level of indirection is eliminated. The code generated by the linearization step would be transformed as follows:

```
i = 0;
while( node != end ) {
  A[i] = *node; /* Do indirection here */
  node = node->Next;
  i++;
}
iMax = i;
for( i = 0; i < iMax; i++ ) {
  /* Result: indirection is eliminated here */
  ... = A[i].Value * B[idx_expr];
}
```

The difference with the code resulting from linearization is that in this case, data is being copied and thus multiple copies of the data from the linked list can exist at run-time. In the restructuring framework presented in the next chapters, copying data is not necessary, if the traversal does not revisit the same node multiple times. In that case, the memory objects can be reordered and all pointers will be updated to reflect this change.

### 4.2.5 Structure Splitting

It is inefficient to copy the entire structure in the initialization loop. Many structure members may not be used in the computation loop and such data would unnecessarily reside in the cache. Additionally, a non-unit stride access pattern on the arrays can prevent other optimizations such as vectorization. By only copying the members which are actually needed, these problems are circumvented. It leads to code that is even easier to analyze. Applied to the code resulting from the indirection elimination step, the following code is obtained:

```
i = 0;
while( node != end ) {
  A_Value[i] = (*node).Value;
  node = node->Next;
  i++;
}
iMax = i;
for( i = 0; i < iMax; i++ ) {
  ... = A_Value[i] * B[idx_expr];
}
```

The concept of structure splitting can also be applied to entire data structures at compile time. This is discussed in detail in Chapter 6.

### 4.2.6 Access Pattern Restructuring

The newly generated computation loop contains an array which is indexed by a new iteration variable ($i$ in this example). Other arrays within the same loop do not follow the access pattern induced by this variable, although the access pattern may

be dependent on the linearized linked list. Consider the computation loop obtained in the previous step. For simplicity, $A\_Value$ is renamed to $A$.

```
for( i = 0; i < iMax; i++ ) {
  ... = A[i] * B[idx_expr];
}
```

This example contains two different access patterns, namely (1) the access pattern induced by $i$ (such a pattern always exists after linearization) and (2) the access pattern induced by $idx\_expr$. In order to impose the same access pattern onto both arrays, either $A$ must be accessed using the access pattern of $B$ or $B$ must be accessed using the access pattern of $A$. Note that the index expression of $B$ may originally have been dependent on the linked list. This expression will have been converted to an array by the indirection elimination and structure splitting step: e.g. $B[A[i] \rightarrow Index]$ would have been transformed to $B[A\_Index[i]]$.

If the array $B$ is remapped using the index expression $idx\_expr$, $idx\_expr$ must be injective (the index expression is viewed as a function of $i$), otherwise at least one element becomes inaccessible. For example, consider the access patterns $(1, 2)$ for $A$ and $(1, 1)$ for $B$. The access pattern of $B$ is not injective and as $A[1]$ can contain only one value, the original semantics of the loop cannot be reproduced.

Consider the two access pattern restructuring techniques, annihilation and sublimation. It is assumed that $A$ is indexed using iteration counter $i$, with lower and upper loop bounds $iMin$ $(= 0)$ and $iMax$, respectively. $B$ is indexed by $idx\_expr$, which is an injective index expression dependent on $i$.

**Annihilation**: Impose the access pattern of $A$ onto $B$, that is, restructure based on the index expression of $A$, which is $i$. This restructuring is done by creating a new array $B'$ which is defined as follows:

$$B'[i] = B[idx\_expr],\ \forall i(iMin \leq i \leq iMax)$$

Note that changing the access pattern of $B$ to follow the access pattern of $A$ is always possible, since the access pattern induced by $i$ is injective. This case is very intuitive: fetch the elements actually needed from the other array $B$ and rewrite the loop such that the restructured array is tightly packed and accessed in the same order as $A$.

**Sublimation**: Impose the (irregular) access pattern of $B$ onto $A$, that is, restructure based on the index expression of $B$, which is $idx\_expr$. This restructuring is done by creating a new array $A'$ which is defined as follows:

$$A'[j] = \begin{cases} A[i] & \text{if } \exists i(idx\_expr(i) = j) \\ identity & \text{otherwise} \end{cases}$$

The variable $j$ ranges from $\min(idx\_expr)$ to $\max(idx\_expr)$, which can be determined in the pre-initialization loop.

In the original loop, not every element of B is necessarily accessed. For instance, if throughout the execution of the loop the variable $idx\_expr$ defines the sequence

$(5, 50, 10)$, then only the elements $A'[5], A'[50]$ and $A'[10]$ need to be defined.

### 4.2.7 Iteration Space Expansion

In case of sublimation, not every element of $B$ (see the previous section) is necessarily accessed. For instance, if throughout the execution of the loop the variable *idx_expr* defines the sequence $(5, 50, 10)$, then only the elements $A'[5], A'[50]$ and $A'[10]$ need to be defined.

However, if the new iteration space is defined to be the interval

$$[\min(idx\_expr), \max(idx\_expr)],$$

the other elements of $A'$ must not alter the semantics of the program. Therefore, the "gaps" in $A'$ should be filled with values chosen in such a way that they do not have any effect when the code is executed. For example, when the loop executes a statement like

$$X = X + A'[i] * Y,$$

then the so called identity value must be 0, as this will preserve the semantics of the program. This case is not really intuitive. The resulting code will most likely have infeasible loop bounds, but this code only serves as an intermediate. The advantage of this approach is that information about the sparse structure is preserved and, together with the input data, a compiler can determine if other access functions can be used, for example storing diagonals of a matrix separately.

### 4.2.8 Loop Extraction

The pre-initialization loop that is generated is placed into same compound statement as the original linked list traversal. Therefore, the initialization loop could end up nested within other loops. In order for the transformations to be efficient, the initialization loop should be extracted from this loop. This transformation enables further optimizations as it often results in a perfectly nested loop. If loop extraction cannot be performed, due to dependencies, another access pattern should be tried in the restructuring phase. This section describes how loop extraction should be performed.

The pre-initialization loop uses a number of variables. Some of these are generated replacement variables which will be used by the transformed main loop. Other variables are used for bookkeeping the pre-initialization loop, such as variables tracking loop boundaries. All remaining variables are non-generated variables or expressions originating from the initial code.

If an initialization loop $A$ is contained in an outer loop $O$, it can be extracted using one of the following two techniques:

1. If all non-generated variables used in $A$ are invariant over $O$, $A$ is just repeating the same operation every iteration of $O$. It is therefore safe to move $A$ in front of $O$ without any further processing.

2. If some non-generated variables used in $A$ are not invariant over $O$, but all non-

invariant variables are only dependent on a loop counter with known bounds, then $A$ can be extracted from $O$.

This requires extension of the generated variables, which is done by adding an extra dimension to these variables, such that for every iteration the correct value is preserved. All references to generated variables must be changed to use the new, extended variables. A new loop $O'$ can be created which uses the original loop structure of $O$. Subsequently, loop $A$ is moved from $O$ to $O'$. Figure 4.3 illustrates variable extension.

(Note that when $A$ is moved in either situation, any statements used to initialize variables used in $A$ will also be moved.)

Loop extraction plays a vital role in the transformation chain. If any dependency prevents loop extraction, the restructuring code and computation code are not separated. In case of annihilation, this results in code where the restructuring is done every iteration, which in itself is expensive and prevents the computation loops to become perfectly nested. In case of sublimation, the lack of separation of restructuring code and computation code prevents recompilation and therefore this optimization path is disabled. An example of a loop containing such a dependency is an algorithm for Conjugate Gradient. This algorithm, which is not discussed here in detail, contains a sparse matrix times dense vector multiplication that is embedded in a loop. It has the following structure:

```
while( ... ) {
  ...
  /* Some definition of vector p */
  ...
  for( row = 0; row < rows; row++ ) {
    /* Linearization code omitted */
    /* Restructuring code */
    for( i = 0; i < iMax; i++ )
      p'[i] = p[ A_ColIndex[i] ];

    for( i = 0; i < iMax; i++ )
      result[row] += A_Value[i] * p'[i];
  }
  ...
}
```

The redefinition of $p$ in each loop renders annihilation inefficient, because the restructuring code cannot be moved in front of the containing while loop. As an alternative, sublimation can be applied (using the access pattern $A\_ColIndex[i]$).

### 4.2.9 Run-time Support for Sublimation

When sublimation is applied, the code obtained from the previous steps (which are all performed at compile-time) results in a code that is never executed. A partial

*Before:*

```
loop O {
  loop A {
    generated-variable =
        original-variable;
   }
   /* additional code including
      transformed main loop
      referencing generated-variable */
}
```

*After:*

```
loop O' {
  loop A {
    generated-variable[E] =
        original-variable;
   }
}
loop O {
   /* additional code including
      transformed main loop
      referencing generated-variable[E] */
}
```

Figure 4.3: Loop $A$ which depends on $E$ is extracted from $O$.

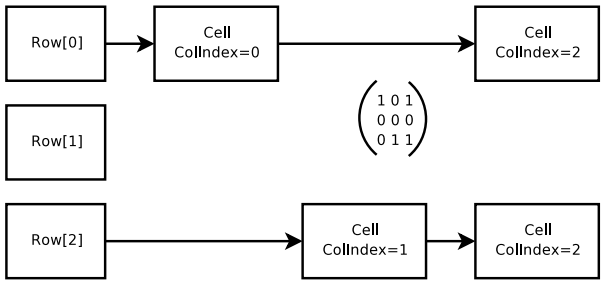



Figure 4.4: A sparse matrix using a linked list representation.

recompilation step is included in the run-time system where the newly generated loops are transformed. Only the computation loops are recompiled while keeping the pre-initialization untouched. During execution of the pre-initialization loop, the non-identity structure of the restructured array is written to a file which is used in the recompilation step to generate instance specific optimized code.

The perfectly nested loop is recompiled using the additional non-identity structure information. The implementation of such a compiler is discussed in depth by Bik and Wijshoff [23, 25, 26]. This recompilation step emits restructured code which is compiled into a shared library which is loaded by the application at run-time. Finally, the newly generated code is executed.

## 4.3 Example

The concept described is this chapter is demonstrated by using a code example of sparse matrix multiplication. Figure 4.4 depicts the data structure used for the representation of a sparse matrix. Figure 4.5 shows the actual C code performing the multiplication. Only the left matrix is sparse (compressed row storage). The right matrix and the result matrix are both dense. The rows of the sparse matrix are traversed using linked lists which prevents optimizations such as loop interchange and vectorization. Additionally, as linked list elements are not generally successive elements in main memory, performance will suffer due to cache misses.

If we consider the loops in the example code, one of these loops (the *while* loop) is a traversal of a linked list. With an $n \times m$ result matrix, this loop is executed $n \cdot m$ times, because the dot product of each row of the left matrix with each column of the right matrix involves the traversal of a linked list which is relatively costly. Ideally, the inner loop would be vectorized, however this is prevented by using the contents of a linked list element as operand for the multiplication ($leftCell \rightarrow Value$) and for indexing of an array ($leftCell \rightarrow ColIndex$).

This code can be transformed to an intermediate code which is equivalent, but uses directly accessed arrays in the inner *while* loop. As an initial step, linearization is applied on the linked list which transforms the linked list traversal into a pre-

```
void MatrixMultiply(Matrix left,
        double **right, double **result,
        int cols )
{
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int row, col;
  for(col=0; col<cols; col++) {
    for(row=0; row<dimensions; row++) {
      leftCell = left.Rows[row];
      while( leftCell != NULL ) {
        result[row][col] +=
            leftCell->Value *
            right[leftCell->ColIndex][col];
        leftCell = leftCell->ColNext;
      }
    }
  }
}
```

Figure 4.5: Sparse matrix multiplication.

initialization loop and a computation loop.

On the resulting code, indirection elimination and structure splitting can be applied. In the computation loop, only the members $Value$ and $ColNext$ are used. These members will be put into an array by applying structure splitting. The code produced by these steps can be found in Figure 4.6. Memory management code is inserted to dynamically resize arrays when needed at run-time. This is necessary, because the length of a linked list is not known a priori.

At this point, two choices can be made: either the access pattern of $A\_Value$ is used to restructure *right* (annihilation) or the access pattern of *right* is used to restructure $A\_Value$ (sublimation). Memory allocation code is left out of these short code samples.

In the first case, *right* is restructured as follows:

```
for( i = 0; i < counter; i++)
  rightp[i] = right[A_ColIndex[i]];
```

*rightp* (*right* prime) is now substituted for *right* in the computation loop:

```
for( i = 0; i < counter; i++ )
  result[row][col] += A_Value[i] * rightp[i][col];
```

In the second case, $A\_Value$ is restructured to follow the access pattern of *right*. This is done by the following loops:

```
for( i = 0; i < MAX_INT; i++ )
  A_Valuep[i] = 0;
for( i = 0; i < counter; i++ )
  A_Valuep[A_ColIndex[i]] = A[i];
```

This code is clearly *not* meant to be executed ever, and the upper bound should be seen as a conceptual upper bound. During recompilation, the correct upper bounds induced by $A\_ColIndex[i]$ can be determined and a feasible iteration space is generated. Now $A\_Valuep$ can be substituted for $A\_Value$:

```
for( i = 0; i < counter; i++ )
  result[row][col] += A_Valuep[A_ColIndex[i]] *
                      right[A_ColIndex[i]][col];
```

In order to remove the indirect addressing from the loop, change the loop control structure such that the loop iterates over the entire range defined by $A\_ColIndex$:

```
for( j = lowerbound; j < upperbound; j++ )
  result[row][col] += A_Valuep[j] *
                      right[j][col];
```

Remember that this does not change the semantics of the program, as the "gaps" in $A\_Valuep$ have been set to 0.

The pre-initialization code depends on the variable *row*, which is not loop-invariant (the containing *for* loop increases it by one every iteration). Therefore, loop extraction cannot be directly applied without eliminating this dependency. Values which are pointed to by $leftCell$, either directly or indirectly (by following a pointer chain), are not modified. Therefore, loop extraction is possible if all variables dependent on $leftCell$ are extended, such that for every iteration of *row*, the values which were copied are stored separately.

The pre-initialization code is now independent of the computation loop and can be moved in front of all containing *for* loops. The resulting code is shown in Figure 4.6, which is code resulting from restructuring $A\_Value$.

For simplicity, we assume a lower bound of 0 on the inner loop. Memory management code is inserted, except for the restructured array $A\_Valuep$. This is not a problem, as this code only serves as an intermediate representation. The code generated in the run-time phase of recompilation will generate the proper memory allocation code.

An interesting issue is the determination of the maximum upper bound of the inner loop. Instead of having a separate upper bound for each row, the maximum upper bound encountered across all iterations is used. This extends the iteration space of the computation loop but poses no further problems, as the compiler knows that any uninitialized element of $A\_Valuep$ is the identity value (in this case 0).

The resulting code only traverses the linked list once per row and the computation loop has become a perfectly nested *for* loop, which is easier to analyze. In this case, loop interchange is enabled so that the inner loop can be vectorized. Additionally,

```
void MatrixMultiply(Matrix left, double **right, double **result, int cols )
{
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int row, col, i;
  double **A_Value = (double **)malloc(sizeof(double *) * dimensions );
  int **A_ColIndex = (int **)malloc(sizeof(int *) * dimensions );
  int *counter = (int *)malloc(sizeof(int) * dimensions );
  for( row = 0; row < dimensions; row++ ) {
    int size = INIT_ALLOC;
    A_Value[row] = (double *)malloc(sizeof(double)*size);
    A_ColIndex[row] = (int *)malloc(sizeof(int)*size);
    leftCell = left.Rows[row];
    counter[row] = 0;
    while( leftCell != NULL ) {
      if( counter == size ) {
        /* Resize array */
        size *= 2;
        A_Value[row] = (double *)realloc( A_Value, sizeof(double) * size );
        A_ColIndex[row] = (int *)realloc( A_ColIndex, sizeof(int) * size );
      }
      /* Code generated by successive application of linearization,
       * indirection elimination and loop extraction (which introduced
       * the extra dimension indexed by row) */
      A_Value[row][counter[row]] = (*leftCell).Value;
      A_ColIndex[row][counter[row]] = (*leftCell).ColIndex;
      leftCell = leftCell->ColNext;
      counter++;
    }
    /* Restructure A_Value - Assume value of undefined element is 0 */
    for( i = 0; i < counter; i++ )
      A_Valuep[row][A_ColIndex[i]] = A[i];
  }
  /* This loop will be recompiled and loaded dynamically */
  for( col = 0; col < cols; col++ ) {
    for( row = 0; row < dimensions; row++ ) {
      for( j = 0; j < MAX_INT; j++ ) {
        result[row][col] += A_Valuep[row][j] * right[j][col];
      }
    }
  }
  for( row = 0; row < dimensions; row++ ) {
    free( A_Value[row] ); free( A_ColIndex[row] );
  }
  free( A_Value ); free( A_ColIndex ); free( counter );
}
```

Figure 4.6: Sparse matrix multiplication after loop extraction (sublimation).

this loop structure dramatically increases cache performance as subsequent items are adjacent in memory.

In this example, the code resulting from restructuring *right* to match the access pattern of $A\_Value$ induced by $i$ (annihilation) can easily be optimized further by performing a simple loop interchange. This would enable vectorization. In this case the run-time recompilation phase is left out, as the iteration space is defined (symbolically) at compile-time. The version resulting from restructuring $A\_Value$ to match the access pattern of *right* induced by $A\_ColIndex[i]$ (sublimation) is not directly executable, as this code would be very inefficient (it has a potentially huge iteration space, as the loop bounds can be anything at run-time). Together with non-identity structure information the computation loop can be transformed to a data instance specific code which can be very efficient, as will be shown in the following section.

## 4.4 Experiments

A prototype C source-to-source restructuring compiler has been implemented that supports the transformations outlined above on simple linked list traversals. The transformation chain has been applied to two benchmarks from the SPARK00 benchmarks (see Chapter 3), SPMATMAT and PCG[1]. SPMATMAT is an example where annihilation is a feasible option. For PCG, annihilation is not successful, as dependencies prevent the cost of restructuring to be amortized over multiple iterations. It is shown that sublimation can be applied in this case, showing significant improvements.

### 4.4.1 Sparse Matrix Times Dense Matrix Multiplication

The transformations described in Section 4.2 have been applied on code performing matrix multiplication, as described in the above example. In this case, the left matrix is a sparse $n \times n$-matrix and the right matrix is a dense $n \times m$-matrix. The result matrix is also dense. Figure 4.5 shows the original algorithm.

All benchmarks have been executed using both the GNU C/Fortran compiler and the Intel C/Fortran compiler. There are five different versions of the program, the original program, two programs generated using sublimation (one uses the Fortran compiler to compile the code emitted by the sparse compiler MT1 [23,25,26] the other uses *f2c* [43], a Fortran to C compiler) and two programs obtained using annihilation (one without further optimizations and one with loop interchange applied). Together, this results into ten different versions.

For sublimation, the access pattern chosen is the access pattern of *right*, which leads to the following computation loop:

[1]Note that throughout this thesis, small differences in the implementation of the benchmarks might exist, as SPARK00 has evolved over time.

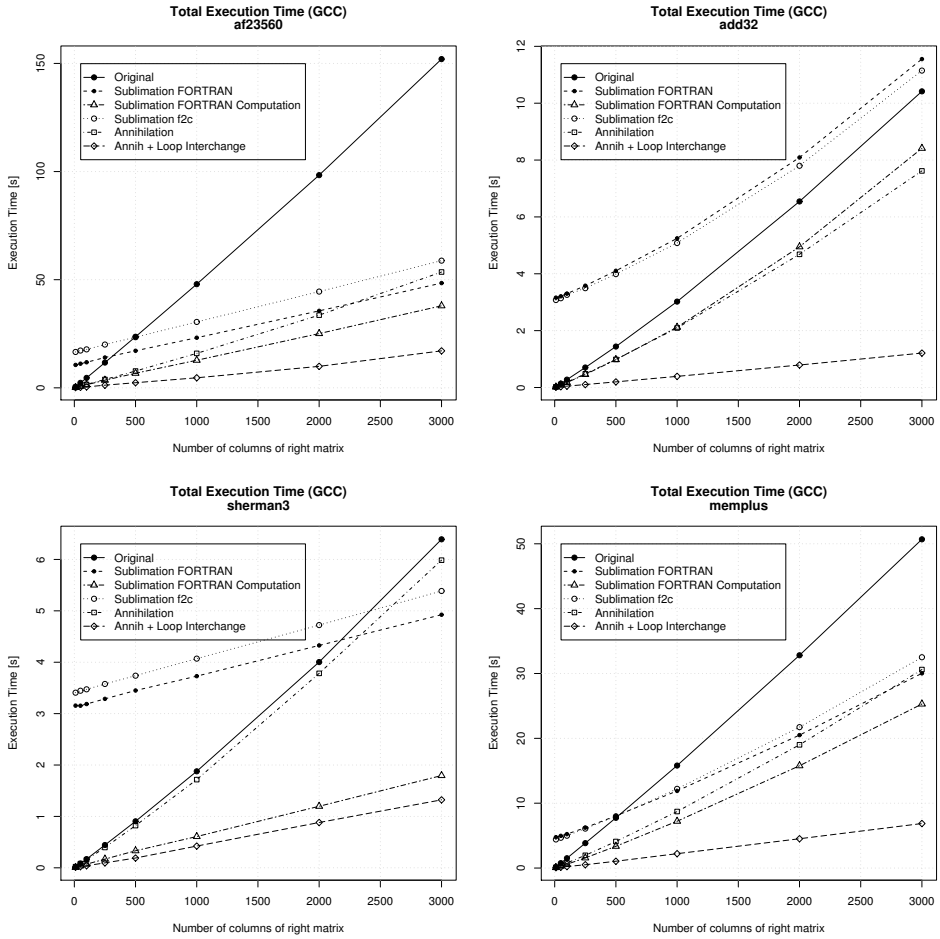



Figure 4.7: Execution times using GNU GCC.

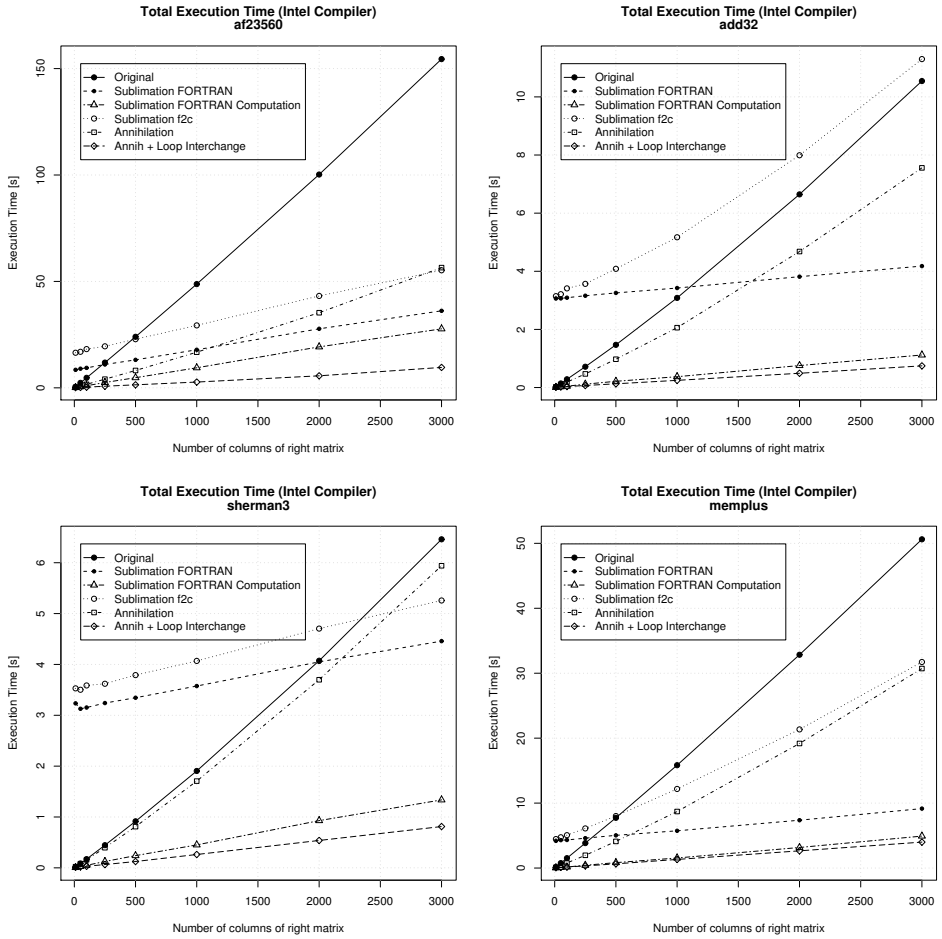



Figure 4.8: Execution times using the Intel Compiler.

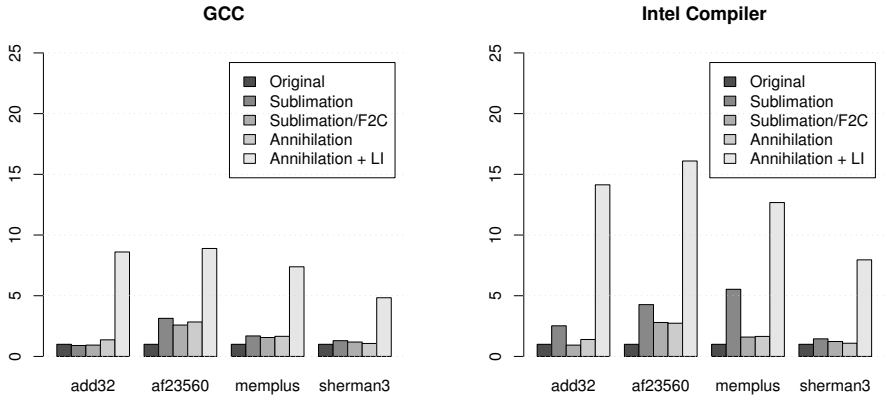



Figure 4.9: Speedup obtained with 3000 result columns.

```
for( col = 0; col < cols; col++ ) {
  for( row = 0; row < dimensions; row++ ) {
    for( i = 0; i < MAX_INT; i++ ) {
      result[row][col] +=
          A_Valuep[row][i] * right[i][col];
    }
  }
}
```

The real bounds of the inner loop are obtained at run-time in the pre-initialization loop. In the example shown, a sparse matrix structure is embedded into a two dimensional array whose size may exceed available memory. As explained in Section 4.2.6, $A\_Value$ is restructured to follow the access pattern of *right*. The *identity* value for the multiplication followed by addition is 0. The access pattern of $A\_Valuep$ is determined in the pre-initialization loop and can be used to restructure the loops such that this structure is taken into account during optimization. Currently, this step is not implemented for C code.

Instead, the intermediate C code is compiled to Fortran. For such code, a restructuring compiler called MT1 exists, which can take this structure into account. This compiler takes array based code using direct addressing and transforms this into an optimized sparse code, taking the non-zero structure of the matrices into account. The non-zero structure must be specified in a separate file. The code emitted in this stage is recompiled at run-time and the generated code is dynamically linked to the application.

In order to do a full comparison of all combinations, the restructured code is compiled, as well as a C version that is obtained by converting the code back to C by the $f2c$ utility. The other versions are generated by applying annihilation, i.e. *right*

is restructured to follow the access pattern of $A\_Value$, which is $i$. This results in the following computation loop:

```
for( col = 0; col < cols; col++ ) {
  for( row = 0; row < dimensions; row++ ) {
    for( i = 0; i < iMax[row]; i++ ) {
      result[row][col] +=
          A_Value[row][i] * rightp[row][i][col];
    }
  }
}
```

After applying loop interchange:

```
for( row = 0; row < dimensions; row++ ) {
  for( i = 0; i < iMax[row]; i++ ) {
    for( col = 0; col < cols; col++ ) {
      result[row][col] +=
          A_Value[row][i] * rightp[row][i][col];
    }
  }
}
```

The compilers used are GCC 4.0.2 and the Intel C and Fortran Compiler 9.1. For GCC, we used the options '-m32 -msse3 -O3 -fomit-frame-pointer', for the Intel Compiler '-msse3 -O3 -fp -fno-alias' (both produce 32-bit executables). The benchmarks were executed on an Intel Xeon 3.00 GHz CPU, 4GB of main memory running SuSE Linux 10.0. Four different matrices which are publicly available [41, 92] were used, namely *sherman3*, *memplus*, *add32* and *af23560*.

Figure 4.7 shows the execution times for sparse matrix times dense matrix multiplication for various sizes of the right matrix using the GCC compiler. Figure 4.8 shows the results for the Intel Compiler. The speedups relative to the original code (GCC code relative to the original code compiled with GCC, Intel Compiler code relative to the original code compiled with the Intel C compiler) when the right matrix has 3000 columns, is shown in Figure 4.9.

As can be seen in Figure 4.7 and Figure 4.8, the run-time recompilation can take a considerable fraction of the total execution time, but it dramatically speeds up the computation itself. In this experiment, the code generated by applying annihilation clearly is the fastest. Table 4.1 shows the initialization time needed by annihilation and sublimation. The execution time of sublimation using Fortran is also plotted using the computation time only (initialization time left out) to indicate the efficiency of the code generated using sublimation. Note that if the initialization time is not taken into account, the code generated by sublimation shows a performance similar to that of annihilation. Especially the Intel Fortran compiler emits code that is close in performance to the code produced by annihilation. This shows that sublimation is a viable alternative if dependencies prevent application of annihilation.

Interestingly, for GCC it does not really matter whether the Fortran or the C version is used. For the Intel Compiler, in two of the four cases it makes quite a

| | | **af23560** | **add32** | **sherman3** | **memplus** |
|---|---|---|---|---|---|
| Sublimation | GCC | 10.439 | 3.130 | 3.124 | 4.688 |
| | Intel | 8.436 | 3.049 | 3.297 | 4.190 |
| Sublimation/F2C | GCC | 16.386 | 3.069 | 3.408 | 4.358 |
| | Intel | 16.344 | 3.146 | 3.482 | 4.419 |
| Annihilation | GCC | 0.073 | 0.006 | 0.005 | 0.029 |
| | Intel | 0.074 | 0.006 | 0.005 | 0.029 |

Table 4.1: Mean initialization time.

difference whether the Fortran or the C version is used. For the matrices *add32* and *memplus* the computation loop is considerably faster if Fortran is used. In principle, the C code emitted by *f2c* should be the same, but apparently the Fortran code is easier to analyze by a compiler. The performance of the C versions is comparable for GCC and the Intel Compiler. Apparently, the Intel Fortran Compiler discovers some parallelism that the GCC Fortran compiler does not find. It can be concluded that sublimation does generate code which is optimizable, but that not every compiler fully exploits these opportunities.

### 4.4.2 Preconditioned Conjugate Gradient

There are cases where annihilation is not a feasible option. Preconditioned conjugate gradient [49] is such an example. This algorithm iteratively solves linear systems. In each iteration, a matrix/vector multiplication is performed. The matrix is constant throughout execution, but the vector changes every iteration. Therefore, restructuring the vector is expensive as it cannot be pulled outside of the containing loop. In addition, this vector is used is subsequent computations which of course expect the original access pattern. In principle, this can be solved by restructuring all dependent data structures, but this has not been implemented yet.

A less disruptive choice in this case is the application of sublimation. Sublimation has been applied to a C implementation of Conjugate Gradient with a diagonal preconditioner. The matrices used are *af23560*, *sherman3*, *memplus*, *impcol_b* and *lshp3466*. The algorithm does not converge, except for *sherman3*. This is not a problem, as we are interested in the program behavior caused by the non-zero structure of the underlying problem, rather than the actual solution.

Table 4.2 shows the execution times of the two programs generated by MTC. One directly compiles the restructured Fortran code, the other uses *f2c* to convert the restructured code back to C. The execution times shown are the average execution times taken over 10 runs. The significance of the difference in execution times is determined using the *Student's t-test*. The threshold used for the *p-value* is 0.001. The only differences found to be significant are for the combination GCC with the matrix *impcol_b*. In the absolute sense, the differences are not very large and therefore the results for the version using *f2c* will not be considered further.

Figure 4.10 shows the execution times of PCG using the different matrices. The

| Matrix | Compiler | Iterations | Total(pcg_mtc) | Total(pcg_mtc_f2c) | p-value |
|---|---|---|---|---|---|
| af23560 | gcc | 0 | 10.4369181 | 10.4565496 | 0.2332 |
| | | 200 | 13.6794500 | 13.6856079 | 0.7277 |
| | | 500 | 18.4899473 | 18.5281476 | 0.0928 |
| | | 1000 | 26.5012467 | 26.5199758 | 0.5196 |
| | icc | 0 | 8.3799467 | 8.3820959 | 0.8543 |
| | | 200 | 11.2647890 | 11.2720330 | 0.4640 |
| | | 500 | 15.6034099 | 15.5978320 | 0.7378 |
| | | 1000 | 22.8165724 | 22.8604394 | 0.3324 |
| impcol_b | gcc | 0 | 2.7630555 | 2.7659491 | 0.7064 |
| | | 1000000 | 7.7020608 | 7.8059651 | 0.0000* |
| | | 2000000 | 12.6517297 | 12.8694533 | 0.0000* |
| | | 5000000 | 27.4455029 | 28.0216110 | 0.0000* |
| | | 10000000 | 52.1405536 | 53.2367686 | 0.0000* |
| | icc | 0 | 2.7902089 | 2.7847962 | 0.7141 |
| | | 1000000 | 5.7294116 | 5.7362139 | 0.1551 |
| | | 2000000 | 8.6936637 | 8.6937868 | 0.9908 |
| | | 5000000 | 17.6339414 | 17.5776914 | 0.5198 |
| | | 10000000 | 32.6823472 | 32.3215858 | 0.1807 |
| lshp3466 | gcc | 0 | 1.9389572 | 1.9393990 | 0.8050 |
| | | 5000 | 3.1645367 | 3.1619932 | 0.9443 |
| | | 10000 | 4.4255588 | 4.4025642 | 0.6569 |
| | | 25000 | 8.2005100 | 8.1453305 | 0.7551 |
| | | 50000 | 14.0215637 | 14.1054011 | 0.7226 |
| | icc | 0 | 1.9219801 | 1.9304343 | 0.0001 |
| | | 5000 | 2.8958285 | 2.9275486 | 0.3830 |
| | | 10000 | 3.9794020 | 3.8251235 | 0.1028 |
| | | 25000 | 6.8309266 | 6.9324833 | 0.6294 |
| | | 50000 | 11.4676299 | 11.6319217 | 0.4393 |
| memplus | gcc | 0 | 3.6738501 | 3.7176453 | 0.1371 |
| | | 200 | 4.9385124 | 4.9246219 | 0.3406 |
| | | 500 | 6.8018818 | 6.8135444 | 0.4001 |
| | | 1000 | 9.9479365 | 9.8877520 | 0.0811 |
| | | 2000 | 16.1671235 | 16.2084638 | 0.4743 |
| | icc | 0 | 3.1837252 | 3.1985074 | 0.6831 |
| | | 200 | 4.1646083 | 4.2815048 | 0.1062 |
| | | 500 | 5.5736581 | 5.5779907 | 0.7269 |
| | | 1000 | 7.9800528 | 7.9539172 | 0.1920 |
| | | 2000 | 12.7488172 | 12.7159817 | 0.5007 |
| sherman3 | gcc | 0 | 3.1321917 | 3.1264834 | 0.7156 |
| | | 200 | 3.2285829 | 3.2185971 | 0.1075 |
| | | 500 | 3.3751111 | 3.3805490 | 0.6407 |
| | | 1000 | 3.6319173 | 3.6358024 | 0.8249 |
| | | 2000 | 4.1335331 | 4.1530528 | 0.7129 |
| | | 2865 | 4.6117277 | 4.6734273 | 0.4756 |
| | icc | 0 | 3.0950571 | 3.0827892 | 0.4930 |
| | | 200 | 3.1625665 | 3.1744411 | 0.2263 |
| | | 500 | 3.3024061 | 3.3019816 | 0.9704 |
| | | 1000 | 3.5116003 | 3.5093471 | 0.9205 |
| | | 2000 | 3.9008675 | 3.8564076 | 0.2961 |
| | | 2865 | 4.2025144 | 4.2786232 | 0.1312 |

Table 4.2: Execution Times for Preconditioned Conjugate Gradient. MTC vs MTC/F2C

results for *lshp3466* are shown separately in Figure 4.11, as for this matrix, the program shows some unexpected behavior. For the matrix *af23560*, the transformations performed by MTC are effective. Using both compilers, significant speedups are obtained. For *memplus*, the transformations are effective as well. PCG converges in 2875 iterations for *sherman3* and at this point the original code performs better. However, note that the slope of the curve is less steep, which means that the computation itself is faster but it cannot compensate for the time spent in the initialization. The results are different for *impcol_b*. In this case, performance suffers if the transformations are applied. This matrix is relatively small (53x53) and no appropriate access pattern can be found for this matrix, which results in the selection of a representation completely based on an indirectly addressed access storage scheme. This storage scheme provides no benefits compared to the linked list representation, which uses indirect addressing as well. The Intel Compiler clearly outperforms GCC on this matrix. For all other matrices in Figure 4.10, the Intel Compiler performs slightly better than GCC.

As mentioned before, PCG shows some abnormal behavior when the matrix *lshp3466* is used. Figure 4.11 shows the execution times for this matrix. In addition to the mean execution time, the 95% confidence intervals are also shown. Both for the Intel Compiler and GCC, there is a large variance in the execution times for the untransformed version. Interestingly, the transformed version does not show this behavior and is likely to execute faster. This indicates that the transformations we propose do not only give gains in performance, but also can result in algorithms that show more robust behavior.

Table 4.3 compares the original algorithm with the version generated by MTC. In all cases (except for the matrix *lshp3466*) the differences in execution times are significant. Table 4.4 compares the performance of the Intel Compiler and GCC. From these data, it can be concluded that the Intel Compiler performs significantly better than GCC.

Speedups for the code produced by MTC have been obtained using the maximum number of iterations that data has been collected for. Additionally, the asymptotic speedup is calculated by leaving out the initialization times. All speedups are compared to the original code, using the same compiler. The results are shown in Figure 4.12. The asymptotic speedup shows that the code produced by MTC is potentially faster than the original code in all cases, except for *impcol_b*.

### 4.4.3 Discussion

The results obtained from the experiments suggest that the transformations proposed in this chapter can be very effective. Code containing pointer chains are hard to analyze and transforming such codes to regularized codes can have major benefits. The methods proposed here result in a code that has the following properties: 1) Loop interchange is enabled and related data exhibits spatial locality due to linearization and structure splitting. 2) Structure splitting reduces the amount of data that must be fetched from memory because if the linked list itself was used in the computation, data that is never needed can be loaded in the CPU cache. 3) Loop extraction results in simpler computation loop structures which are easier to analyze. 4) After

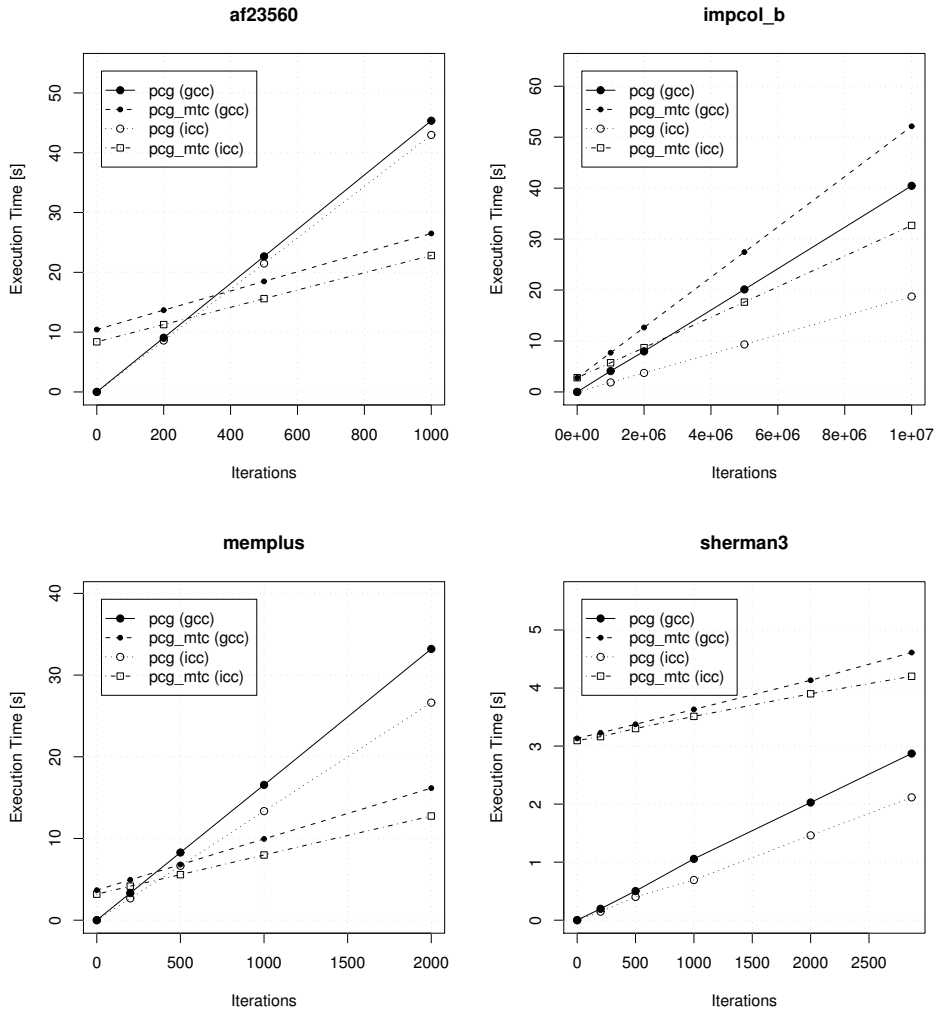



Figure 4.10: Execution Times for Preconditioned Conjugate Gradient

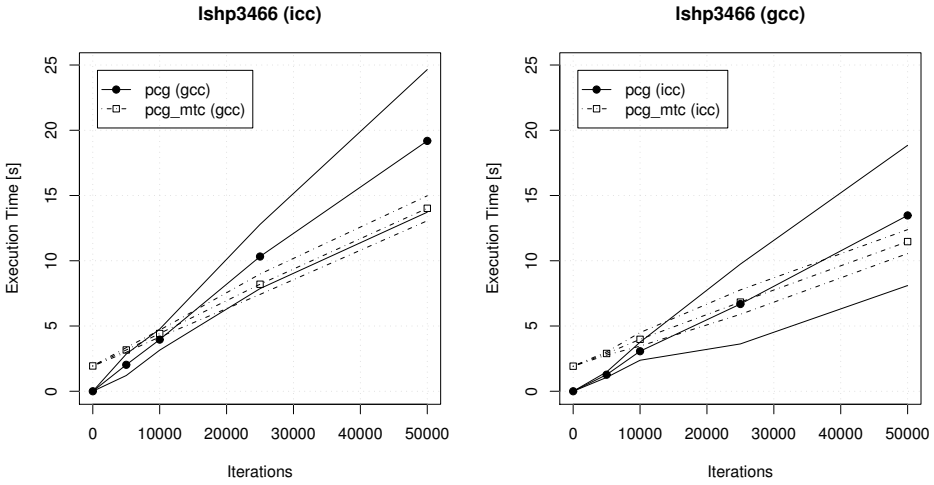



Figure 4.11: Execution Times for Preconditioned Conjugate Gradient (lshp3466)

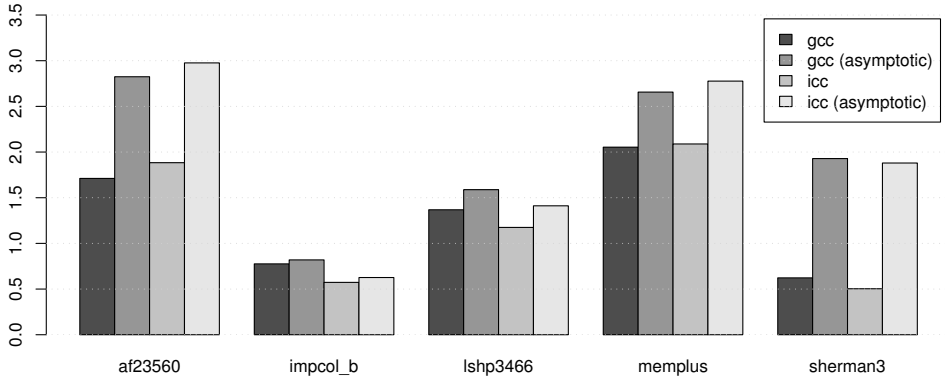



Figure 4.12: Mean Asymptotic Speedup Obtained by Using MTC

| Matrix | Compiler | Iterations | Total(pcg) | Total(pcg_mtc) | p-value |
|---|---|---|---|---|---|
| af23560 | gcc | 0 | 0.0014977 | 10.4369181 | 0.0000* |
| | | 200 | 9.0676140 | 13.6794500 | 0.0000* |
| | | 500 | 22.6633046 | 18.4899473 | 0.0000* |
| | | 1000 | 45.3536471 | 26.5012467 | 0.0000* |
| | icc | 0 | 0.0011748 | 8.3799467 | 0.0000* |
| | | 200 | 8.6016974 | 11.2647890 | 0.0000* |
| | | 500 | 21.4728851 | 15.6034099 | 0.0000* |
| | | 1000 | 42.9741284 | 22.8165724 | 0.0000* |
| impcol_b | gcc | 0 | 0.0000489 | 2.7630555 | 0.0000* |
| | | 1000000 | 4.1009672 | 7.7020608 | 0.0000* |
| | | 2000000 | 7.9636513 | 12.6517297 | 0.0000* |
| | | 5000000 | 20.1213715 | 27.4455029 | 0.0000* |
| | | 10000000 | 40.4568902 | 52.1405536 | 0.0000* |
| | icc | 0 | 0.0000546 | 2.7902089 | 0.0000* |
| | | 1000000 | 1.8692082 | 5.7294116 | 0.0000* |
| | | 2000000 | 3.7324763 | 8.6936637 | 0.0000* |
| | | 5000000 | 9.3331125 | 17.6339414 | 0.0000* |
| | | 10000000 | 18.7334097 | 32.6823472 | 0.0000* |
| lshp3466 | gcc | 0 | 0.0002808 | 1.9389572 | 0.0000* |
| | | 5000 | 2.0282342 | 3.1645367 | 0.0000* |
| | | 10000 | 3.9577893 | 4.4255588 | 0.0058 |
| | | 25000 | 10.3251169 | 8.2005100 | 0.0004 |
| | | 50000 | 19.1854449 | 14.0215637 | 0.0002 |
| | icc | 0 | 0.0002627 | 1.9219801 | 0.0000* |
| | | 5000 | 1.2696215 | 2.8958285 | 0.0000* |
| | | 10000 | 3.0687554 | 3.9794020 | 0.0000* |
| | | 25000 | 6.6853483 | 6.8309266 | 0.7832 |
| | | 50000 | 13.4729768 | 11.4676299 | 0.0470 |
| memplus | gcc | 0 | 0.0011539 | 3.6738501 | 0.0000* |
| | | 200 | 3.3156697 | 4.9385124 | 0.0000* |
| | | 500 | 8.2835206 | 6.8018818 | 0.0000* |
| | | 1000 | 16.5571594 | 9.9479365 | 0.0000* |
| | | 2000 | 33.1987006 | 16.1671235 | 0.0000* |
| | icc | 0 | 0.0009095 | 3.1837252 | 0.0000* |
| | | 200 | 2.6729744 | 4.1646083 | 0.0000* |
| | | 500 | 6.6520123 | 5.5736581 | 0.0000* |
| | | 1000 | 13.3496854 | 7.9800528 | 0.0000* |
| | | 2000 | 26.6269574 | 12.7488172 | 0.0000* |
| sherman3 | gcc | 0 | 0.0003997 | 3.1321917 | 0.0000* |
| | | 200 | 0.1960706 | 3.2285829 | 0.0000* |
| | | 500 | 0.5022527 | 3.3751111 | 0.0000* |
| | | 1000 | 1.0568317 | 3.6319173 | 0.0000* |
| | | 2000 | 2.0277288 | 4.1335331 | 0.0000* |
| | | 2865 | 2.8716864 | 4.6117277 | 0.0000* |
| | icc | 0 | 0.0003540 | 3.0950571 | 0.0000* |
| | | 200 | 0.1463178 | 3.1625665 | 0.0000* |
| | | 500 | 0.4011187 | 3.3024061 | 0.0000* |
| | | 1000 | 0.6927279 | 3.5116003 | 0.0000* |
| | | 2000 | 1.4612195 | 3.9008675 | 0.0000* |
| | | 2865 | 2.1161639 | 4.2025144 | 0.0000* |

Table 4.3: Execution Times for Preconditioned Conjugate Gradient. Original vs. MTC

| Matrix | Program | Iterations | Total(ICC) | Total(GCC) | p-value |
|---|---|---|---|---|---|
| af23560 | pcg | 0 | 0.0011748 | 0.0014977 | 0.00000* |
| | pcg | 200 | 8.6016974 | 9.0676140 | 0.00000* |
| | pcg | 500 | 21.4728851 | 22.6633046 | 0.00000* |
| | pcg | 1000 | 42.9741284 | 45.3536471 | 0.00000* |
| | pcg_mtc | 0 | 8.3799467 | 10.4369181 | 0.00000* |
| | pcg_mtc | 200 | 11.2647890 | 13.6794500 | 0.00000* |
| | pcg_mtc | 500 | 15.6034099 | 18.4899473 | 0.00000* |
| | pcg_mtc | 1000 | 22.8165724 | 26.5012467 | 0.00000* |
| impcol_b | pcg | 0 | 0.0000546 | 0.0000489 | 0.00977 |
| | pcg | 1000000 | 1.8692082 | 4.1009672 | 0.00000* |
| | pcg | 2000000 | 3.7324763 | 7.9636513 | 0.00000* |
| | pcg | 5000000 | 9.3331125 | 20.1213715 | 0.00000* |
| | pcg | 10000000 | 18.7334097 | 40.4568902 | 0.00000* |
| | pcg_mtc | 0 | 2.7902089 | 2.7630555 | 0.08049 |
| | pcg_mtc | 1000000 | 5.7294116 | 7.7020608 | 0.00000* |
| | pcg_mtc | 2000000 | 8.6936637 | 12.6517297 | 0.00000* |
| | pcg_mtc | 5000000 | 17.6339414 | 27.4455029 | 0.00000* |
| | pcg_mtc | 10000000 | 32.6823472 | 52.1405536 | 0.00000* |
| lshp3466 | pcg | 0 | 0.0002627 | 0.0002808 | 0.00001* |
| | pcg | 5000 | 1.2696215 | 2.0282342 | 0.00023 |
| | pcg | 10000 | 3.0687554 | 3.9577893 | 0.00007* |
| | pcg | 25000 | 6.6853483 | 10.3251169 | 0.00002* |
| | pcg | 50000 | 13.4729768 | 19.1854449 | 0.00021 |
| | pcg_mtc | 0 | 1.9219801 | 1.9389572 | 0.00000* |
| | pcg_mtc | 5000 | 2.8958285 | 3.1645367 | 0.00000* |
| | pcg_mtc | 10000 | 3.9794020 | 4.4255588 | 0.00035 |
| | pcg_mtc | 25000 | 6.8309266 | 8.2005100 | 0.00000* |
| | pcg_mtc | 50000 | 11.4676299 | 14.0215637 | 0.00000* |
| memplus | pcg | 0 | 0.0009095 | 0.0011539 | 0.00000* |
| | pcg | 200 | 2.6729744 | 3.3156697 | 0.00000* |
| | pcg | 500 | 6.6520123 | 8.2835206 | 0.00000* |
| | pcg | 1000 | 13.3496854 | 16.5571594 | 0.00000* |
| | pcg | 2000 | 26.6269574 | 33.1987006 | 0.00000* |
| | pcg_mtc | 0 | 3.1837252 | 3.6738501 | 0.00000* |
| | pcg_mtc | 200 | 4.1646083 | 4.9385124 | 0.00000* |
| | pcg_mtc | 500 | 5.5736581 | 6.8018818 | 0.00000* |
| | pcg_mtc | 1000 | 7.9800528 | 9.9479365 | 0.00000* |
| | pcg_mtc | 2000 | 12.7488172 | 16.1671235 | 0.00000* |
| sherman3 | pcg | 0 | 0.0003540 | 0.0003997 | 0.00000* |
| | pcg | 200 | 0.1463178 | 0.1960706 | 0.00000* |
| | pcg | 500 | 0.4011187 | 0.5022527 | 0.00000* |
| | pcg | 1000 | 0.6927279 | 1.0568317 | 0.00000* |
| | pcg | 2000 | 1.4612195 | 2.0277288 | 0.00000* |
| | pcg | 2865 | 2.1161639 | 2.8716864 | 0.00000* |
| | pcg_mtc | 0 | 3.0950571 | 3.1321917 | 0.11104 |
| | pcg_mtc | 200 | 3.1625665 | 3.2285829 | 0.00000* |
| | pcg_mtc | 500 | 3.3024061 | 3.3751111 | 0.00000* |
| | pcg_mtc | 1000 | 3.5116003 | 3.6319173 | 0.00006* |
| | pcg_mtc | 2000 | 3.9008675 | 4.1335331 | 0.00050 |
| | pcg_mtc | 2865 | 4.2025144 | 4.6117277 | 0.00008* |

Table 4.4: Execution Times for Preconditioned Conjugate Gradient. Intel Compiler vs. GCC

loop extraction the expensive traversal of a linked list is less deeply nested, removing many unnecessary traversals of the linked list. Together, these properties enable major performance improvements.

The choice between annihilation and sublimation involves a tradeoff. Annihilation is relatively simple and does not need run-time support, sublimation is more complicated and has significant run-time overhead. Table 4.1 shows how large this difference can be. Sometimes a choice is forced as dependencies prevent one of the two methods to be applied, as was shown in the example in Section 4.2.8. If there are no constraints and both methods are available, the experiments suggest that annihilation is the way to go. We believe however that there are cases where sublimation is superior to annihilation, because in principle, sublimation allows to choose different basis functions to access the iteration space. In that case, at run-time data structures can be created to match this new basis.

## 4.5 Summary

Although it is well known that the choice of data structures has a big influence on the performance of algorithms, most compiler transformations do not directly target data structures but only consider the computational code. One of the reasons for this is that especially pointer structures impose severe restrictions on standard transformations. Various transformations have been described in this chapter that directly target the transformation of data structures and it is shown that substantial speedups can be achieved.

The framework outlined in this chapter has been implemented in a prototype restructuring compiler (called MTC) and extending this to a production quality compiler is not a trivial task. Many requirements must be met before these concepts are widely applicable. In the subsequent chapters, different facets of the concepts outlined here will be dealt with in more detail, and more generic techniques to handle a much wider class of applications are described. One must realize that before a restructuring framework has described here can be fully functional, a large implementation effort is needed. For full applications, the problems which need to be dealt with are the propagation of information on data access restructuring, injectiveness of indirect addressing, and the structure of the underlying data structure. In the remainder of this thesis, the LLVM compiler framework will be used together with Data Structure Analysis [72], which is used to determine how data is accessed throughout an application.

Actually, the restructuring concepts described in this chapter might be seen as a first step in realizing data structure independent programming. In [71,119] and [120], proposals have been made to define future programming where specific information on how data is stored and structured is hidden from the core application. More generically phrased: Can the component based software architecture be seen as a way to compartmentalize specific implementation details of the building blocks and can the data virtualization approach be seen as a specific instance of this component based approach.

# CHAPTER 5

# LLVM Preliminaries

The previous chapter sketched out the main ideas for restructuring pointer-linked data structures and code. All examples were C-to-C restructuring transformations. The C language has many constructs that need to be handled and therefore normalization techniques are required in order to simplify the analysis phase. Necula et al. define CIL [89], a type-safe subset of C, which simplifies C to a kind of simpler normal form. While this approach is suitable for C-to-C restructuring, it does not widen the scope of compiler techniques to other languages.

Compiler intermediate codes also provide a much simpler set of constructs. For example, GCC has its GIMPLE intermediate [1], which is based on the SIMPLE intermediate used in the McCat compiler [52]. This is a C-like three address code [10]. LLVM is a relatively new compiler framework [74], which operates on the LLVM intermediate representation, referred to as LLVM bitcode. The main advantage of LLVM over CIL is that it is, in principle, language independent. For any front-end emitting LLVM bitcode, the optimization framework can be applied. Compared to GCC, LLVM is easier to work with and an analysis of fundamental importance for our work is available, namely Lattner's Data Structure Analysis [72]. Therefore, LLVM is the compiler framework that will be used in the remainder of this thesis.

A number of issues are solved by using the LLVM compiler framework. First of all, it offers a clean and simple instruction set for which optimizations can be defined. The bitcode provides an implicit normalization of C code (and other languages for which front-ends exist). The indirection elimination and structure splitting techniques explained in the previous chapter can be implemented in a more transparent way. This will be explained in Section 5.4. Furthermore, in the subsequent chapters, restructuring techniques that substitute indirection elimination, loop expansion and loop extraction, that use a run-time system, will be discussed.

In this chapter, we will briefly describe the LLVM compiler framework, and some analysis and transformation passes on which our restructuring transformations, that are presented in the subsequent chapters, depend. Section 5.1 describes the LLVM framework and bitcode in general. Data Structure Analysis (DSA) is an analysis pass developed by Lattner [72] that provides information about data structure usage and safety. It is described in Section 5.2. Using DSA, Lattner implemented automatic pool allocation of disjoint data structures [73, 75]. On top of this, program-wide structure splitting can be implemented. These techniques are explained in Section 5.2 and 5.4, respectively.

## 5.1 The LLVM Compiler Infrastructure

The LLVM compiler infrastructure started as a project at the University of Illinois. Lattner and Adve characterize the goal of the framework as follows [74]:

> "LLVM is designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs."

The LLVM compiler tries to achieve this by compiling different languages into a single, intermediate representation, the LLVM bitcode. For example, each separate C, C++, Fortran or Ada file can be compiled into an LLVM bitcode module. These modules can then be linked by the LLVM linker, which results in a single, linked bitcode module. This flexibility allows for optimization at link-time, which enables more aggressive optimizations to be applied.

In this section, we will briefly describe the LLVM bitcode, and one important analysis, Data Stucture Analysis, that partitions data structures within a whole-application and identifies how data structures are accessed throughout the application.

### LLVM Bitcode

LLVM bitcode consists of a virtual instruction set with an unbounded amount of virtual registers and a main memory that can be read and written. The virtual registers are only assigned once, so at the virtual register level, LLVM bitcode uses a static single assignment (SSA) representation. In addition, all register, load and store instruction are typed. Thus, LLVM bitcode is a typed language, which is quite close to machine languages, but has type information associated with each instruction. Let us consider an example piece of LLVM bitcode:

```
bb:                                                ; preds = %bb1
  %0 = call i32 @rand() nounwind                   ; <i32> [#uses=1]
  %1 = sitofp i32 %0 to double                     ; <double> [#uses=1]
  %2 = fdiv double %1, 0x41747AE140000000          ; <double> [#uses=1]
  %3 = fsub double %2, 5.000000e+01                ; <double> [#uses=1]
  %4 = load double** %vec_addr, align 8            ; <double*> [#uses=1]
  %5 = load i32* %i, align 4                       ; <i32> [#uses=1]
  %6 = sext i32 %5 to i64                          ; <i64> [#uses=1]
  %7 = getelementptr double* %4, i64 %6            ; <double*> [#uses=1]
  store double %3, double* %7, align 1
  %8 = load i32* %i, align 4                       ; <i32> [#uses=1]
  %9 = add i32 %8, 1                               ; <i32> [#uses=1]
  store i32 %9, i32* %i, align 4
  br label %bb1
```

In this code, the variables %0 through %9 are SSA register variables. It is possible to give SSA variables a name, but that is not shown in this example. Text following a semicolon are comments. The first instruction is a *call* instruction, that calls the `rand` function, which is a standard C library function. The result is returned into %0. As this is an SSA variable, we know that %0 will only be assigned here, and nowhere else. We also know that any uses of this SSA variable must be dominated by the definition of this variable. The *sitofp* instruction used the value resulting from the call, and converts it to a double precision floating point number. Note, that for each instruction, the full type of the arguments and the return type is known. Any type conversions are stated explicitly.

Accessing memory is done through the load and store instructions. The *load* instruction takes a typed pointer, and returns the value that is read from memory. The *store* instruction takes both a pointer and a value, and stores the value at the location specified by the pointer. The store instruction does not return any value. Again, note that the operands and return value are all typed variables.

### Address Calculations

Address arithmetic is implemented using a special instruction, *getelementptr*, which preserves all type information. In the example above,

```
%7 = getelementptr double* %4, i64 %6
```

corresponds to the C expression:

```
var7 = &var4[var6]; // or var7 = var4 + var6;
```

It is crucial to note that the *getelementptr* instruction *never* dereferences a pointer. It only performs pointer arithmetic. The *getelementptr* instruction is also used to calculate pointers to fields in structures. For example, the corresponding LLVM data type for *struct timeval* from the C library is:

```
%struct.timeval = type { i64, i32 }
```

If `%t` is a pointer referring to a variable of this type, then the address of the second field, with type *i32*, is computed using the following *getelementptr* instruction:

```
%fieldAddr = getelementptr %struct.timeval * %p, i32 0, i32 1
```

In terms of C, this translates to:

```
fieldAddr = &p[0].field1;
```

This means that starting from pointer `%p`, we need the element 0 (`%p` is interpreted as an array) and within this element, we need the address to field 1 (which is the second field).

Of crucial importance in this thesis is the possibility of link-time optimization. With link-time optimization, a whole-program view is provided, which is needed to prove safe use of data structures. This is determined by Lattner's Data Structure Analysis, which is discussed in the next section. More details on the LLVM compiler infrastructure and the bit code instructions can be found on the LLVM project website [7].

## 5.2 Data Structure Analysis

Lattner's Data Structure Analysis (further referred to as DSA) is an efficient, interprocedural (whole program), context- and field-sensitive pointer analysis [72, 73, 75]. It provides information about how data structure elements are actually accessed in a program and (conservatively) identifies disjoint instances of data structures, even if these data structures show an overlap in the functions that operate on them. Such disjoint data structures can be allocated in their own designated memory area, called a memory pool. First, it is important to understand that DSA is *not a shape analysis*. DSA determines which data structures can be proved to be disjoint in memory. Such a data structure can be a linked list, a tree, a graph or any other pointer-linked data structure.

The result of DSA is the Data Structure Graph (DS Graph). Within this graph, the nodes represent memory objects. A node is described as follows [72]:

> "Each DS graph node represents a (potentially unbounded) set of dynamic memory objects and distinct nodes represent disjoint sets of objects, i.e., the graph is a finite, static partitioning of the memory objects. Because we use a unification-based approach, all dynamic objects which may be pointed to by a single static pointer variable or field (in some context) are represented as a single node in the graph."

Our primary interest lies in the nodes that are type-homogeneous, that is, all memory objects represented by the node are of the same type *and* are used in a type-consistent way throughout the entire program. For implementation and efficiency reasons, data structures should often not be stored according to their layout as defined by the

programmer, as this layout is not optimal, and is merely chosen because of, for instance, its logical structure. As DSA provides information on type-safety on the whole-program level, it is possible to remap the layout of data structures. This assumes that all uses of such a data structure have been identified and that the data structure cannot escape the program (otherwise it would not be type-safe). These restructuring transformations will be discussed in the next chapters. Here, we will focus on the framework supporting such transformations.

Construction of the DS graph is done in three phases. The first is the *Local Analysis Phase*, during which the actual program representation is used to construct DS graphs for all functions, taking only local information into account. For each function, each data access is analyzed and each SSA variable that contains a pointer will have an edge to the corresponding data structure it is pointing to in the graph. The layout is determined by examining how data is accessed. For example, consider the following code:

```
%struct.MatrixElement = type { double, %struct.MatrixElement*,
                               i32, i32, %struct.MatrixElement* }
...
%17 = load %struct.MatrixElement** %pElement, align 8
%18 = getelementptr %struct.MatrixElement* %17, i32 0, i32 0
%19 = load double* %18, align 8
%20 = load %struct.MatrixElement** %pElement, align 8
%21 = getelementptr %struct.MatrixElement* %20, i32 0, i32 2
%22 = load i32* %21, align 8
```

Variable %17 will contain the pointer loaded from *pElement*. We do not consider the code preceding this example, so we cannot determine the effect of this load statement. Let us assume that in the DS graph, %17 has an edge pointing to some data structure node. The next *getelementptr* instruction computes an offset relative to %17. This offset is 0. Note, though, that this computes a pointer to the first field of the structure that %17 is pointing to, which has the type *double*. The subsequent load accesses this location as a *double*. Thus, in the data structure graph, the node that %17 is connected to will record that it is accessed at offset 0 as a double. The next *getelementptr* performs a similar operation, but computes a pointer to the third field, which is an integer. Thus, the layout of the DS node looks as follows now, represented as a sequence of offset/type pairs: `<<0:double,8:void,8:i32>>`.
If now, for some reason, this same data structure would be accessed in a different way (due to type-unsafe programming, or inheritance in object oriented languages), a conflict may arise. Imagine that the third field is accessed as a 64-bit integer. In that case, a conflict with a previously recorded type is detected, and the node is marked as type-unsafe (*collapsed*, in DSA terminology).

DS nodes contain flags which indicate whether they contain complete information, that is, whether information from callees and callers has been integrated in the node for the current function. The nodes also indicate whether data associated with a node might be read from (R flag) or written to (M flag). After the first phase, this indicates whether the data is read or written in the function that is associated with the node.

```
int main( int argc, char **argv )
{
    ...
    MatrixPtr tmp = ReadMatrixPtrRow( matrixFile );
    MatrixPtr Matrix = MatrixToFormat( tmp, format );
    ...
    for( i = 0; i < iterations; i++ )
        MatrixMultiplyVec( Matrix, right, result );
    ...
}
```

Figure 5.1: Code excerpt of main function of SPMATVEC.

The next phase, the *Bottom-Up Analysis Phase*, combines the information from the local functions with results from their callees, by propagating this information bottom-up. This phase is context-sensitive (that is, it takes function calls into account). After this phase, the DS nodes represent information about the aggregate effect of the function itself and all its potential callees. This requires a mapping between DS nodes in callers and callees. After this mapping has been made (which can be found by mapping actual arguments to formal arguments), the information in the nodes is merged. If type conflicts are detected, the node is marked unsafe in the caller, but not in the callee. This is because the callee still might be type-safe if called from another context.

The last phase is the *Top-Down Analysis Phase*, which propagates information on data structure usage from callers to callees. This provides a global, conservative view on data structures in the application. If at some point it is determined that a data structure is type-unsafe, this is propagated throughout the application after the Top-Down Analysis Phase. We will not need the Top-Down Analysis Phase in our restructuring framework, the results from the Bottom-Up Analysis suffice.

Let us illustrate this with an example. Figure 5.1 shows a part of the main function of SPMATVEC, one of the SPARK00 benchmarks described in Chapter 3. Figure 5.2 shows the associated DSGraph. Information about the variables generated by the compilation to the LLVM bit code (which uses an SSA representation) is not shown. The graph shows the two stack variables (specified by the $S$ flag) $\%tmp$ and $\%Matrix$, which both have their disjoint storage space on the stack. Hence the separate nodes. The $MatrixFrame$ structure they are both pointing to is one node, indicating that the analysis cannot prove that they are pointing to disjoint structures. The $MatrixFrame$ structure basically contains three pointers. These are the three arrays of pointers that point to the start of a row, the start of a column and the diagonal elements. The $MatrixElement$ structure is the structure containing the matrix data. It has two self references, which are the two pointers used to traverse the matrix row- and column-wise.

Each function has its own bottom-up DS graph. Nodes that are related to formal arguments are data structures that are passed in by calling the function. Nodes that

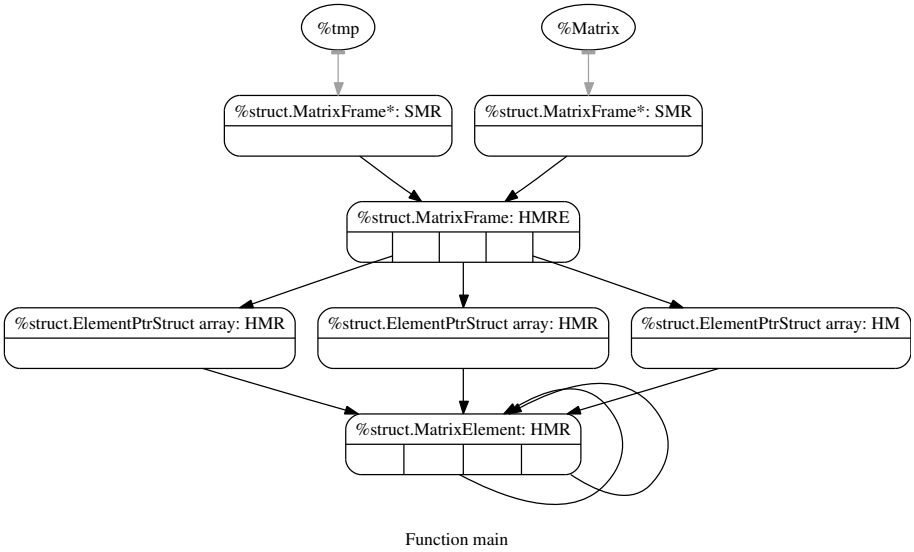



Figure 5.2: DSGraph for main function of SPMATVEC benchmark.

do not correspond to a formal argument depict data structures that are instantiated within this function. At this point, such a node incorporates all information on how this node is used in all callees. The Bottom-Up Analysis ensures that if a node is used in a type-safe fashion this is propagated to the point where the data structure in instantiated. At this particular location, a choice can be made how this data structure instance is treated.

Summarizing, for each data structure, we are interested at which point it is actually instantiated and whether it is type-safe in all callees. All such data structures can be stored in a disjoint memory segment, called a memory pool.

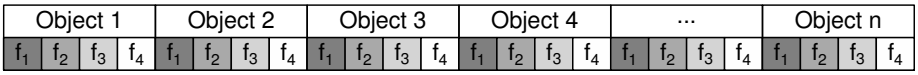
## 5.3 Automatic Pool Allocation

On top of DSA, Lattner et al. implemented *automatic pool allocation* [73, 75]. Pool allocation is a transformation which rewrites calls to memory allocation functions to custom memory allocators such that disjoint data structures are allocated from disjoint memory regions. This is done by identifying pool-allocatable data structures, as shown in the previous section. As such a node in the DS graph has been determined to be type-safe, all associated memory allocation functions can be identified and rewritten such that they call a pool allocation library. The new allocation functions take an additional argument compared to their standard equivalents, the *pool descriptor*, which uniquely identifies a data structure instance at run-time. Pool-allocated structures that are allocated from type-homogeneous pools, allow for precise control of the

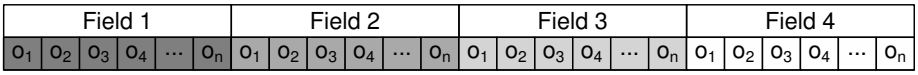
```
struct S {
    int32 x;
    double y;
    struct S *next;
} [SIZE_OF_POOL];
```

```
struct S {
    int32 x[SIZE_OF_POOL];
    double y[SIZE_OF_POOL];
    struct S *next[SIZE_OF_POOL];
};
```

Figure 5.3: Array of structures vs. structure of arrays.




(a) Regular type-homogeneous pool




(b) Split type-homogeneous pool

Figure 5.4: Type-homogeneous pools can be stored either grouped by object or by field. In both cases, locations are uniquely identified by the triplet $(pool, object, field)$.

data layout, because it is known that all allocated elements within a particular memory region have the same type. We use this property to modify the way structures are laid out in main memory.

## 5.4 Pool-Assisted Structure Splitting

A useful data layout transformation when a data structure is known to be type-safe is *structure splitting*. Let us consider a memory pool that only stores elements of a particular structured type. In essence, such a pool is just an array of structures (AOS). If we assume that the size of this array is fixed, this can be easily transformed into a structure of arrays. Figure 5.3 depicts this concept by giving the corresponding structure definitions in C. As the size of all fields is known and we also know that only objects of a single type exist within a pool, data can be addressed by the triplet $(pool, object, field)$. This is a logical addressing mode, and the underlying physical mapping can be picked freely. Figure 5.4 shows how this logical addressing mode is mapped to both a regularly ordered pool and a split pool.

Splitting structures has some advantages over normal pool allocation. Firstly, it is possible to do away with all padding which is otherwise needed (except for alignment-imposed restrictions) as primitive data types (i.e. floats, doubles, integers etc..) normally must be aligned to addresses corresponding with the size of the type. In a split structure however, the elements that follow each other will be of the same type and size, this means that the fields can be packed much more efficiently in many cases where padding would normally be inserted. Secondly, another advantage is that a field in a structure that is not accessed as often as the other elements will not pollute the cache, as unused data will not be taking up cache space.

Structure splitting has its limitations, for example, a split structure will typically be split over multiple memory pages and thus require more active TLB[1] entries. As a consequence of this, a structure that is not used in sequential access (e.g. by following pointer chains), is not likely to yield any performance benefits when split. In addition, when multiple fields of a structure are referenced, the cache efficiency will be worse for split structures than for a non-split structure as in the split version, as the two fields will be located in two different cache lines, whereas in the original version, those fields are most likely co-located in the same cache line.

The implementation of our structure splitting transformation is similar to the DSA-based implementation of Curial et al. [35], who implemented structure splitting in the IBM XL compiler. Compared to their implementation, we have improved on the efficiency of address calculations. Furthermore, we have extended DSA MOD flags, which indicate whether a data structure might be modified in a particular function. The standard MOD-flags are compile-time flags. Our implementation supports field-sensitive, run-time MOD-flags which are set whenever a write to a particular field occurs.

Using the tools described in this chapter, we will describe the implementation of our restructuring framework for pointer-linked data structures.

[1] Translation Lookaside Buffer

# CHAPTER 6

# A Compilation Framework for Automatic Restructuring

Predictability in memory reference sequences is a key requirement for obtaining high performance on applications using pointer-linked data structures. This contradicts the dynamic nature of such data structures, as pointer-linked data structures are often used to represent data that dynamically changes over time. Also, different traversal orders of data structures cause radical differences in behavior. Thus, having control on data layout is essential for getting high performance.

For example, architectures like the IBM Cell and GPU architectures each have their own characteristics and if algorithms using pointer-structures are to be executed on such architectures, the programmer must mold the data structure in a suitable form. For each new architecture, this means rewriting code over and over again.

Another common pattern in code using pointer-linked data structures is the use of custom memory allocators. Drawbacks of this approach are that such allocators must be implemented for various problem domains and that they depend on the knowledge of the programmer, not on the actual behavior of the program. Our restructuring framework is a first step in the direction to liberate the programmer from having to deal with domain specific memory allocation and rewriting of data structures.

## 6.1 Outline

In this chapter, we present a compiler transformation chain that determines a type-safe subset of the application and enables *run-time* restructuring of type-safe pointer-linked data structures. This transformation chain consists of type-safety analysis after which disjoint data structures can be allocated from separate memory pools. At run-time, accesses to the memory pools are traced temporarily, in order to gather actual memory access patterns. Next, from these access patterns, a permutation is

generated which enables the memory pool to be reordered. Note that these traces are not fed back into a compiler, but are rather used to restructure data layout at *run-time* without any modification of the original application. Pointers in the heap and on the stack are rewritten if the target they are pointing to has been relocated. After restructuring, the program resumes execution using a new data layout.

Restructuring of linked data structures cannot be performed unless a type-safe subset of an application is determined. This information is provided by Lattner and Adve's Data Structure Analysis (DSA), which has been explained in Chapter 5. The result of DSA can be used to segment disjoint data structures into different memory regions, the memory pools. Often, many memory pools turn out to be type-homogeneous, that is, they only store data of a specific (structured) type.

For type-homogeneous pools, we have implemented *structure splitting*, similar to MPADS [35], the memory-pooling-assisted data splitting framework by Curial et al. This changes the physical layout of the structures, but logically they are still addressed in the same way (any data access can be characterized by a *pool, objectid and field* triplet). Structure splitting is not a strict requirement for restructuring, but it simplifies the implementation and results in higher performance after restructuring.

In order to restructure, a permutation vector must be supplied. This permutation vector is obtained by tracing memory pool accesses. Tracing does have a significant impact on performance, so in our framework tracing can be disabled after a memory pool has been restructured. The application itself does not need to be aware of this process at all. While in principle such a trace can also be used to modify the behavior of a memory allocator for the next execution of an application, we have not done so at this moment. It is important to note that tracing and restructuring all happen within *a single run* of an application.

Section 6.2 describes the compile-time parts of our framework and Section 6.3 treats the run-time components. Section 6.4 contains the experimental evaluation of our framework. Restructuring pointer-linked data structures has great potential and considerable speedups are shown on the SPARK00 benchmarks. The challenge of SPARK00 lies in closing the performance gap between pointer traversals resulting in random access behavior and traversals resulting in perfectly sequential access behavior. As such, it illustrates the potential, but it does not guarantee that such speedups will be obtained for any application. The overhead of the tracing mechanism, which of course does not come for free, is discussed in Section 6.4.2. It is shown that the performance gains do compensate for this overhead within relatively few consecutive uses of the restructured data structure. Restructuring memory pools requires a special stack that can be updated after restructuring. Different mechanisms and their implications are discussed and evaluated in Section 6.4.3. Address calculations need to be efficient, and therefore we present improved address calculations (compared to the address calculations as given by Curial [35]) for addressing fields of structures in split memory pools in Section 6.4.4. A summary is given in Section 6.5.

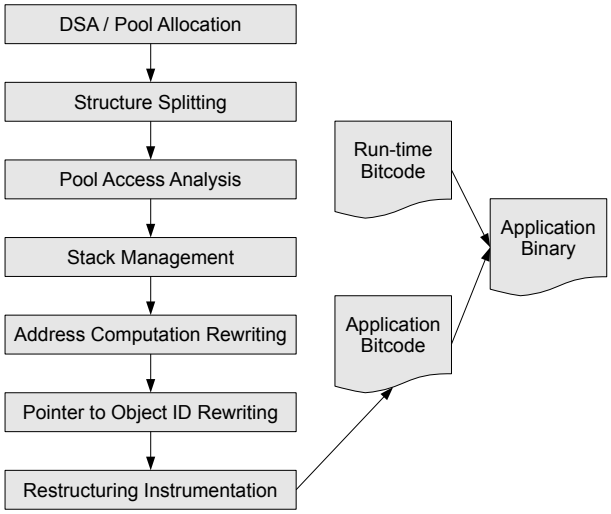



Figure 6.1: Overview of the pool restructuring compilation chain.

## 6.2 Compile-time Analysis and Transformation

At compile-time, a whole program transformation is applied in order to obtain a split structure layout that supports run-time restructuring. Figure 6.1 shows an overview of the entire compilation chain that our framework consists of. In this section, the analyses and rewriting compiler passes are discussed.

### 6.2.1 Structure Splitting

Our analysis and transformation chain starts at the point where DSA has been performed on a whole program and pool allocatable data structures have been determined. We start at the *main* function and traverse all reachable functions, cloning each function which needs to be rewritten to support the data layout of split structures. Functions are cloned as there might also be calling contexts in which splitting cannot be applied, and these cases must also be dealt with correctly. For the identification of the memory pools we depend on Lattner and Adve's Pool Allocation [73,75]. It is not possible to split pools that are not type homogeneous, as addressing of object fields would become ambiguous and fields of different types and length would introduce aliasing of field values. This information is however available from the DSA and pool allocation passes.

During the analysis phase, function clones are generated for split versions of functions and calls are rewritten accordingly. Rewriting of other instructions, such as

address calculations are deferred to a later stage, as they are nothing more than a change in the semantics of the address calculation instruction (*GetElementPtrInst*) in LLVM.

Various pieces of information are gathered in the structure splitting analysis pass which are used in subsequent passes. All loads and stores to pool data are identified as well as all loads and stores that store a pointer into a pool, which is needed to support the use of *object identifiers* instead of pointers (see Section 6.2.5). The structure splitting pass ensures that all the address calculation expressions (*GetElementPtrInst* in LLVM) whose result points to data in split pools are identified. These expressions must be rewritten before the final code generation at a later stage. The address calculation expressions are not rewritten immediately but just before code generation because additional passes will need to reason about these expressions.

### 6.2.2 Pool Access Analysis

Pool access analysis is a pass in which all pool accesses (loads and stores) are analyzed. The result of the analysis is that instead of being viewed as an access using a specific pointer, the location read from or written to is represented using a triplet *(pool, object, field)*. *Pool* is the pool descriptor used at run-time, *object* the pointer to the object the data belongs to and *field* is the field number that is accessed. Originally, a load or store just used a pointer as its address operand, but now, the more generic notion of pool, object and field can be used. This is analogous to data access in a database (table, row and column).

For each load and store from a split pool, the analysis is performed as follows:

```
// Get accessed object
baseObject = get underlying object for accessed object
check that baseObject is also a load

pool = get pool descriptor associated with baseObject

// Get accessed field
gepiInst = get pointer operand of memory instruction
check gepiInst is a GetElementPtr instruction
field = get field index from gepiInst
```

Note that for each access to a pool, it must be possible to determine which field is accessed. This property cannot always be proved if the address of fields is taken, and therefore we do not allow that *any address* of a field is written to *any memory* location using the LLVM StoreInst. For example, the following C-code snippet will never be restructured:

```
obj->ptr = &p1->y;
    ...
*obj->ptr = val;
```

This might be a bit over-conservative, and in a future version, we might define this

| Method | Advantages | Disadvantages |
|---|---|---|
| *Pointer Tracking* | Simple<br>Portable | Slow<br>Interferes with IR |
| *Shadow Stack* | Fast<br>Portable | Interferes with IR |
| *Stack Map* | Fast<br>No IR Interference | Backend Modifications<br>Stack walking not portable |

Table 6.1: The three stack management options and their individual advantages and drawbacks

more precisely. Lattner and Adve's pointer compression applies the same restriction on field accesses [76].

### 6.2.3 Stack Management

The primary requirement for structure splitting to work (in terms of code modifications) is the remapping of address calculation expressions so that data is read and written to the relocated location in the split pool. However, if reordering of the pool contents is to be accomplished this is not sufficient. Other pools may for example contain references to the reordered pool (which means that those references need to be updated). However, these on-heap pointers are not the only references to pool objects that the system needs to deal with. The other type of references that need to be managed are pointers that are stored on the stack and that point into the pool. This problem is similar to what garbage collectors have to do, and in their terminology, the on-stack pointers are known as roots. Tracking the on-heap pointers can be done by adding additional meta data to the pool descriptor, this meta data is derived from the DSA (that keeps track of connectivity information between pools).

Three different alternatives to accurate stack managing were explored and evaluated. These approaches include *explicit pointer tracking*, *shadow stacks* and *stack maps*. However, only the first method was fully implemented for reasons that will become clear later on. The three different investigated methods for stack management are summarized in Table 6.1.

#### Explicit Pointer Tracking

One approach to the stack root issue, is to ensure that all pointers are explicitly tracked at the LLVM level. We call this technique pointer tracking. When a pool descriptor is allocated, a special segment of data is acquired that will be used to track all stack local pointers pointing into the pool, whenever a pointer is allocated on the stack, the location of this pointer is inserted in the per pool stack tracking block. A frame marker is needed in order to be able to remove all the pointer tracking entries associated with a returning function. In LLVM, this means that any pointer that is an SSA register must explicitly be stored on the stack. The following LLVM function

illustrates this:

```
void @func(pooldesc *pool0) {
entry:
bb0:
  %x = load {i32, i32}** %heapObjectAddr
  call void @foo %x
  ret
}
```

The function listed above is then transformed into the following:

```
void @func(pooldesc *pool0) {
entry:
  %xptr = alloca {i32, i32}**
  call void @split_st_reg_stack_obj %pool0, %xptr
  call void @split_st_push_frame %pool0
bb0:
  %x = load {i32, i32}* %heapObjectAddr
  store %x, %xptr
  %x_foo_arg = load {i32, i32}* %xptr
  call void @foo %x_foo_arg
  call void @split_st_pop_frame %pool0
  ret
}
```

In the transformed function the pointer *%x* is explicitly backed by a stack variable and this variable is then registered with the run-time function *split_st_reg_stack_obj*. After the pointer registrations a call to the run-time function *split_st_push_frame* is executed. This function will close the stack frame for the current function in order to speed up the pop operation of the stack. It should be noted that the run-time functions mentioned here are very short (a few instructions) and will be inlined. Thus, they do not induce any function calling overhead. In order to reduce this overhead, an approach where stack tracking is disabled in certain functions has been chosen. The pseudo code in the following example illustrates why this is useful:

```
Pool pool;
Matrix *mtx = readMatrix(pool);

doMatrixOperation(pool, mtx);

PoolRestructure(pool, mtx);

for (int i = 1 ; i < N ; i ++) {
 doMatrixOperation(pool, mtx);
}
```

Here the critical code is the *doMatrixOperation*, but if this operation does not call the *PoolRestructure* function, then this function does not need to track the pointers.

Explicit pointer tracking is relatively easy to implement. In terms of implementation effort it is small compared to the methods described further on in Sections 6.2.3 and 6.2.3. Although the method (as shown in experiments later on) is not a good choice for a field deployment, it takes very little code to implement both the passes and the run-time support for the explicit pointer tracking.

#### Shadow Stacks

The second approach that we investigated for tracking pointers on the stack, was the utilization of a shadow stack. This technique is based on the garbage-collection method described by Henderson [51]. Shadow stacks are generated by the compiler. For each function, a data structure is generated that stores pointers to all the pointers that will be used within the function. When a function is called, such a structure is allocated on the stack and this structure is then registered with the run-time. Compared to the pointer tracking method, this reduces the number of calls to the run-time system to register the shadow stack frame to exactly one call per function call as only one pointer (to the shadow stack) needs to be registered.

#### Stack Maps

The third alternative is the construction of stack maps [9]. Stack maps are structures that are generated statically for each function; these structures describe the stack frames of the corresponding functions. The maps are computed during the code generation phase and contain information on for example frame pointer offsets of the pointers allocated by the function. The main advantage of delaying this to the code generation phase is that the transformation will not interact in any way with earlier optimizations. The main drawback is that the stack walking will become platform dependent and this may not necessarily suit every compiler.

### 6.2.4 In-Pool Addressing Expression Rewriting

For non-split structures, the derived pointers to fields in the structures can easily be computed by adding a constant offset to the base pointer of the structure. For split structures however, this is not possible anymore. In a split structure, the field addresses do no longer have constant offsets from the base pointer of the structure (see Figure 6.2 for a graphical explanation of why this happens).

It is obvious that calculating addresses for the fields in the structures must be very efficient. This fact was already stressed by Curial, but he did not optimize the address calculation expressions and their selection rules to the same extent as we did. If this calculation is inefficient, it potentially nullifies much of the performance improvement gained from the more cache-efficient split structure representation. In general, the offset for field $n$ can be represented by the following equation:

$$offset_n = k_n + sizeof_n \frac{p\&(sizeof_{pool} - 1)}{sizeof_0} - p\&(sizeof_{pool} - 1) \tag{6.1}$$

where $k_n$ is the constant offset to field array $n$ from the pool base. The sub-expression $p\&(sizeof_{pool} - 1)$ calculates the object pointer $p$'s offset from the pool base and the expression $\frac{p\&(sizeof_{pool}-1)}{sizeof_0}$ calculates the object index in the pool[1].

Note that when the accessed field is the first field of the structure then $offset_0 = 0$ and if the size of the accessed field is the same as the first field of the structure then $offset_n = k_n$.

We have observed that in many common cases the size difference between the accessed field and the first field is a power of two. Taking this observation into account, we introduce two additional expressions. When the size of the accessed field is greater than the first field of the structure we have that:

$$offset_n = k_n + (sizeof_n - sizeof_0)\frac{p\&(sizeof_{pool} - 1)}{sizeof_0} \tag{6.2}$$

and when the size of the first field is greater than the accessed field use the following expression:

$$offset_n = k_n - (sizeof_0 - sizeof_n)\frac{p\&(sizeof_{pool} - 1)}{sizeof_0} \tag{6.3}$$

Equation 6.1 can be viewed as adding the pool base to the offset from the address of the $n^{th}$ field of the first object, see Figure 6.2. This figure also demonstrates that Equation 6.2 and 6.3 take into account the linear drift of field $n$ due to the size differences between fields 0 and $n$, with respect to the object's pool index and the constant offset $k_n$.

It is assumed that further passes of the compiler will apply strength reduction on all multiply and divide instructions involving a power of two constant. Fog [44] gives the cost for various instructions for a $45nm$ Intel Core 2 CPU. These numbers have been used to estimate the cost in cycles for the various equations calculating the offsets. Assuming that the expressions have been simplified as much as possible through, for example, constant folding and evaluation, we get that when $sizeof_0$ is a power of two, Equation 6.1 will take 26 cycles, if $sizeof_n$ is not a power of two the same equation will take 6 cycles and if both sizes are a power of two it will take 4 cycles. Equation 6.3 will take 3 cycles, and Equation 6.2 will take 3 cycles in the normal case (or 2 cycles if $sizeof_n - sizeof_0 = sizeof_0$).

The address calculations as defined by Equations 6.1 and the elimination of calculations if accessing the first field are already used in MPADS [35], but our additional Equations 6.2 and 6.3, have some important properties. They allow the calculation of the field offsets to be reduced to 2 or 3 instructions instead of 4, as the code generator will merge the divide and the multiplication operation into a single shift operation. Note that for Equation 6.2 when $sizeof_n - sizeof_0 = sizeof_0$, LLVM will automatically eliminate the multiply and the divide instruction, giving even more savings.

The most notable equation cost (26 cycles) has a divide instruction in the expres-

[1]Note that in this context, & is the C-operator for a *bitwise AND*.

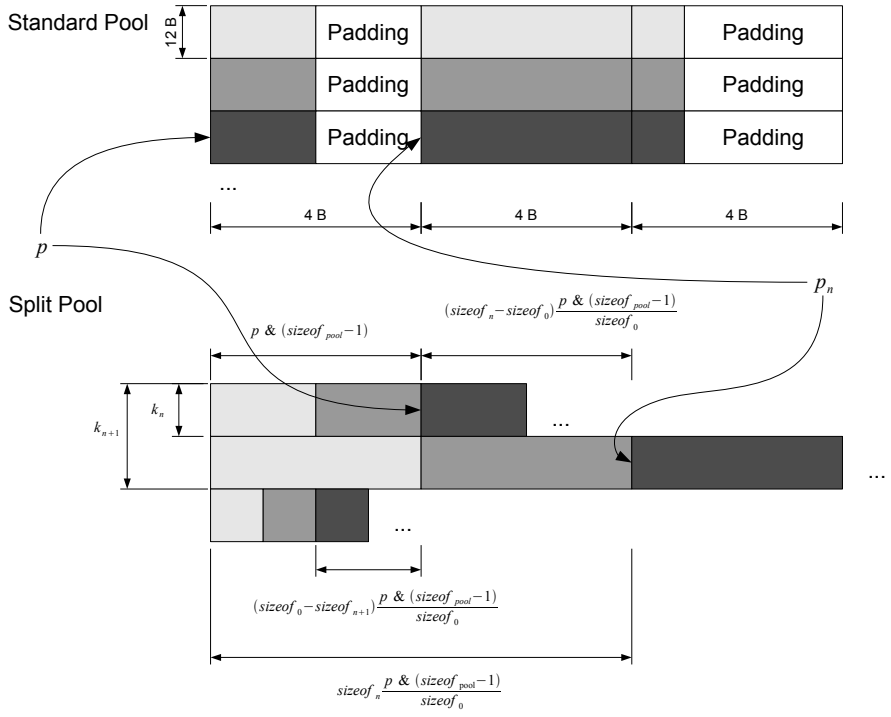



Figure 6.2: Graphical representation of split pools and the field offset expressions detailed in Section 6.2.4 for a split pool consisting of structures with elements of sizes 2, 4 and 1 bytes. Each shade of gray represents an individual object. The object pointer $p$ is in this case is pointing at the third object and the derived pointer $p_n$ is pointing to the second field of the third object.

sion. This will for example happen when the first field of a structure is an array of three 32 bit values (arrays are not split since they are already sequential) and the next element is a 32 or 64 bit value. In those cases up to 23 cycles may be saved on the address calculation as the divide instruction has been eliminated through strength reduction possibilities introduced by Equation 6.3.

Overall it can be said that a compiler that splits structures should also reorder the fields in a structure so that address calculations are made as simple as possible. For example, if a structure contains three fields of lengths 1, 2 and 4 bytes, then the field ordering should place the 2-byte element first, under the condition that the access frequency of the fields is the same, as this simplifies the address calculations. Putting the 2-byte element first will simplify Equation 6.2, (because $sizeof_n - sizeof_0 = sizeof_0$ when accessing the 4-byte field). The size difference with the other field is 1, which is a power of two. As a result, Equation 6.3 can use a shift instruction instead

of a division. At this moment our implementation does not reorder fields.

### 6.2.5 Converting Between Pointers and Object Identifiers

Instead of storing pointers in split memory pools, object identifiers are used. Object identifiers can be used in type-homogeneous pools to uniquely identify an object within a pool. As shown in Section 6.2.2, together with a field number, each data element can be addressed. Object identifiers are a more compact representation than pointers and also more compact than byte offsets from a pool base pointer, as used in Lattner and Adve's *static pointer compression* [76]. Object identifiers are more compact than byte offsets because object identifiers can use every available integer for addressing, while for byte offsets, the next available offset is the previous offset plus the size of one element. Their *dynamic pointer compression* transformation also uses object identifiers. In that case, it provides a representation independent of the size of fields, whereas byte offsets would need to be rewritten if field sizes change.

Our motivation to use objects identifiers is different. While our framework would also benefit from pointer compression (currently object identifiers are stored as 64-bit unsigned integers), we use object identifiers because they can be used as indices in permutation vectors and because they provide position independence for data structures. For future developments, the object indexing will aid in using data structures in hybrid architectures and environments, as the representation is position-independent. Whenever a pointer is loaded from memory, the conversion is done in an architecture and context-dependent way.

Section 6.2.2 described how all loads and stores to memory pools can be represented as a *(pool, object, field)* triplet. In the case that *field* is a field that is pointing to pool allocated data (whether this defines a recursive data structure or a link to another data structure does not matter), this value needs to be converted to an object identifier upon a store, and must be converted to a pointer upon a load. Loads and stores to the stack are unaffected and thus will contain real pointers.

For stores the value to store is rewritten as follows:

```
uintptr_t ptr_to_objid(split_pooldesc_t *pool, void *obj)
{
  if( obj == 0 ) return 0;  // Special case: NULL pointer
  else {
    uintptr_t poolBase = (uintptr_t)pool->data;
    uintptr_t objOffset = poolBase - (uintptr_t)obj;
    uintptr_t objIdx = objOffset / sizeof_field(0);
  }
  return objIdx;
}
```

And for loads the loaded value is rewritten as follows:

```
void *objid_to_ptr(split_pooldesc_t *pool, uint64_t *objIdx)
{
  if( objIdx == 0 ) return 0; // Special case: NULL pointer
  else {
    uintptr_t poolBase = (uintptr_t)pool->data;
    uintptr_t objOffset = objIdx * sizeof_field(0);
    uintptr_t obj = poolBase + objOffset;
    return (void *)objIdx;
  }
}
```

Note that the actual implementation uses LLVM bit code and uses a bitmask instead of an if-statement to handle the NULL pointer.

Compared to the description of object indexing used in the pointer compression transformation by Lattner and Adve [76], our implementation differs in some ways. In their work, object indices are not only present in the heap, but are also used on the stack and in LLVM's virtual registers. Pointer comparisons and assignments do not need the object identifier to be expanded to a full pointer in their framework. In our framework, only loads and stores of pointers (only to pool objects) to split pools need rewriting, and the rest of the code will run unchanged. It also simplifies the restructuring step: on the heap, we only need to handle object identifiers, on the stack we only have to deal with full pointers.

### 6.2.6 Restructuring Instrumentation

Pool tracing and restructuring of data structures requires instrumentation of the code with calls to the tracing run-time. During pool access analysis, all loads and stores to pools have been identified and are represented using the triplet *(pool, object, field)*. All these instructions can be instrumented such that a per pool, per field trace of object identifiers is recorded. Currently, we only trace load instructions.

As tracing is a method that does not come for free, we only enable tracing for one execution of a function and its callees. After the first execution, the data is restructured and tracing is disabled. This is accomplished by generating two versions of the function, one with and one without tracing. Selecting the proper function is done through a global function pointer which is set to the non-traced version after a trace has been obtained.

## 6.3 Run-time Support

Extracting a type-safe subset of the program and replacing its memory allocation by a split-pool-based implementation requires run-time support, similar to the run-time support provided for regular pool allocation. The split pool run-time provides create and destroy functions for split pools as well as memory allocation and deallocation functionality. In addition, some common operations implemented in the standard C library are also provided, such as *memcpy* (which needs to copy data from multiple regions due to the split layout), thereby widening the applicability of the framework.

In this section, the run-time system for splitting and restructuring is described. While this run-time system has been implemented specifically to support our pool restructuring framework, it can also be used as a standalone library, giving the user the ability to explicitly use split and restructurable data structures. Note that pool connectivity must be explicitly specified if the library is used separately from the compiler in order to keep data structures consistent after restructuring.

### 6.3.1 Application Programming Interface

The split-pool run-time offers implementations for initializing pools and for memory allocation and deallocation. Table 6.2 describes the run-time functions needed to support restructuring of split pools.

### 6.3.2 Tracing and Permutation Vector Generation

In order to restructure a memory pool, a permutation must be supplied to the restructuring run-time. The pool access analysis pass (Section 6.2.2) provides the compile-time information *(pool, object and field)* about all memory references and these memory references can all be traced. Separate traces are generated for each field within a pool. For each pool/field combination, this results in a trace of object identifiers. From any of these traces, a permutation vector can be derived which can be used to permute a pool. The permutation vector is currently computed by scanning the trace sequentially and appending the encountered object identifiers to the vector, avoiding duplicates:

```
perm[0] = 0;
permLen = 1;
for (i = 0; i < maxTraceEntry; i++) {
  if ( !perm[trace[i]] ) {
    perm[trace[i]] = permLen;
    permLen++;
  }
}
```

Element 0 is reserved to represent the NULL pointer and therefore is never permuted.

For the evaluation of our restructuring method we choose to trace the first execution of a specified function (a compiler option specifies which function). Subsequently, we restructure the data structure used in this function using this trace and then, upon the next function call, we call the specified function with tracing disabled. In a future implementation, this will be dynamic and tracing could be triggered if a decrease in performance is detected (for example by using hardware counters).

### 6.3.3 Pool Reordering

One of the more important parts of our system is the pool rewriting support. Rewriting in this context means that a pool is reordered in memory, so that it is placed in a

| Function | Arguments | Return value | Description |
|---|---|---|---|
| split_pool_vargs | • split_pooldesc_t *pool<br>• uintptr_t obj_cnt<br>• uint32_t unsplit_obj_len<br>• uint32_t split_obj_len<br>• uint32_t field_cnt<br>• ... | void | **Split pool creation and initialization.** Initializes a new pool, and reserves memory for *obj_cnt* number of objects. Non-split objects length and split object length are both specified. *field_cnt* specifies the number of fields and is followed by a list of integers specifying the size of each field in bytes. |
| split_pooldestroy | • split_pooldesc_t *pool | void | **Destroys a pool and frees up all memory mapped regions.** |
| split_poolalloc | • split_pooldesc_t *pool<br>• unsigned NumBytes | void * | **Allocate Memory from a Split Pool.** Allocates an integer number of objects from a pool. The number of objects allocated is *NumBytes*, divided by the non-split object length, which was specified upon initialization of the pool. This a replacement for *malloc*. |
| split_poolrealloc | • split_pooldesc_t *pool<br>• void *obj<br>• unsigned NumBytes | void * | **Reallocate Memory from a Split Pool.** Replacement for *realloc* in the standard C library. |
| split_poolfree | • split_pooldesc_t *pool<br>• void *obj | void | **Free Pool Allocated Objects.** Replacement for *free* in the standard C library. |
| split_pooltrace_init | • split_pooldesc_t *pool | split_pooltrace_info * | **Initialize tracing for a pool.** Initializes tracing data structures for a pool. |
| split_pooltrace_trace_base_stack<br>split_pooltrace_trace_base_heap | • void *pool<br>• uint32_t field<br>• void *ptr | void | **Adds an entry to the trace for a specific field of a pool.** Two versions exist, one for pointers on (not to!) the stack that are dereferenced, one for pointers on the heap. This is done because pointers on the heap are stored as object identifiers and pointers on the stack are stored as full pointers and thus need to be converted to an object id first when adding an entry to a trace. |

Table 6.2: Functions of the split pool run-time with restructuring support.

hopefully more optimal way with respect to memory access sequences. This is done during run-time, and the re-writing is based on passing in a permutation vector generated during run-time as described in Section 6.3.2. We have implemented a copying rewriting-system that uses permutation vectors that specify the new memory order of the pool. Although permutation vectors could technically be generated during compile time in some cases where data is not input dependent, this has not been seen as necessary at this point in time.

When a permutation vector is available, a pool can be rewritten in order to optimize the memory layout. The pool rewriting algorithm that we have devised has three distinct phases:

1. Pool rewrite, where the actual pool-objects are being reordered

2. Referring pool rewrite, where pointers in other pools that refer to the rewritten pool are updated to the new locations

3. Stack update, where the on-stack references to objects in the rewritten pool are updated

The basic algorithm for the interior pool update is as follows:

```
newData = mmap(pool.size);
foreach field in pool {
  foreach element in field {
    if field contains recursive pointers {
      newData[field][permVec[element]]
       = permVec[pool.data[field][element]];
    } else {
      newData[field][permVec[element]]
       = pool.data[field][element];
    }
  }
}
munmap(pool.data);
pool.data = newData;
```

In this case each field in the split pool is copied into the new address space, and relocated according to the permutation specified in the permutation vector. If the value in the field is itself a pointer to another object in the pool, that pointer is remapped to its new value.

For the second phase where all the referring pools are updated, the rewrite is even simpler:

```
foreach referrer in pool.referrers {
  foreach entry in referrer.field {
    referrer.field[entry] = permVec[referrer.field[entry]];
  }
}
```

Here, each pool that refers to the rewritten pool will have the field containing those pointers updated with the new locations. The algorithm assumes that each pool descriptor has information available regarding the pool connectivity (that is, it knows which fields in other pools are pointing to objects in the rewritten pool). This information can be derived from the DSA discussed earlier. This connectivity information is therefore registered as soon as the pool is created.

### 6.3.4 Stack Rewriting

As already discussed in Section 6.2.3, the program stack is managed through explicit pointer tracking. When a pool descriptor is allocated, a special segment of data is acquired that is used to track all pointers on the stack pointing into the pool. Whenever a pointer is allocated on the stack, the location of this pointer is inserted in the per pool stack tracking block.

When a pool is rewritten, the current stack will be traversed and all base and derived pointers to locations within the pool are rewritten to reflect the new location of the object. This block makes a distinction between base pointers and derived pointers, and each derived pointer is also tagged (in the stack tracking block) with the field it is referring to.

## 6.4 Experiments

The challenge of a restructuring compiler is to generate code that will automatically restructure data, either at compile- or at run-time, in order to achieve the performance that matches the performance when an optimal layout would be used. In the introduction the potential of restructuring was shown by comparing execution of the benchmarks using explicitly defined data layouts. Ideally, we want to obtain similar performance gains, but then by automatic restructuring of the data layout of the used pointer-linked data structures.

The benchmark set SPARK00 (see Chapter 3), which contains pointer benchmarks, is used. In these benchmarks, the layout can be controlled precisely. The pointer-based benchmarks used are: SPMATVEC (sparse matrix times vector), SPMATMAT (sparse matrix times matrix), DSOLVE (direct solver using forward and backward substitution), PCG (preconditioned conjugate gradient) and JACIT (Jacobi iteration).

These benchmarks store their matrix using orthogonal linked lists (elements are linked row-wise and column-wise). All of them traverse the matrix row-wise, except DSOLVE, which traverses the lower triangle row-wise and the upper triangle column-wise.

For all benchmarks, one iteration of the kernel is traced, after which the data layout is restructured. After this, tracing is disabled. This all happens at run-time, without any hand-modifications of the application itself.

The experiments have been run on two platforms. The first is the Intel Core 2 platform, an Intel Xeon E5420 2.5 GHz processor with 32 $GiB$ of main memory,

running Debian 4.0. The other system is an Intel Core i7 920 2.67GHz based system with 6 $GiB$ of main memory, running Ubuntu 9.04.

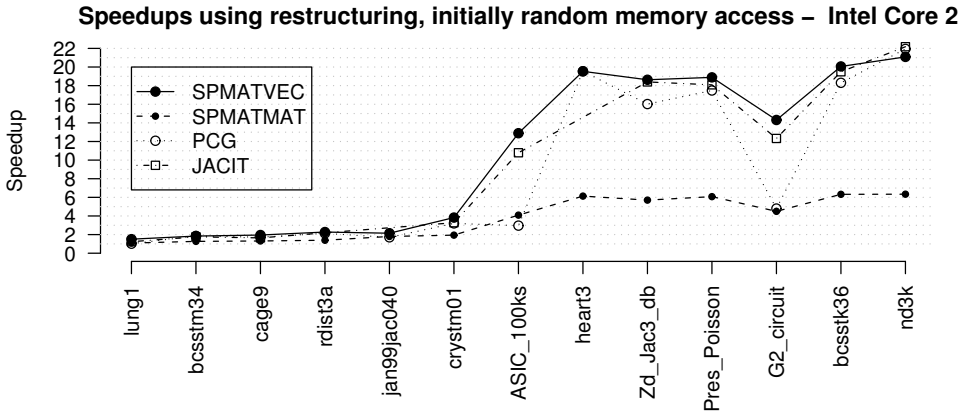
### 6.4.1 Pool Reordering

As shown in the introduction, being able to switch to an alternative data layout can be very beneficial. We applied our restructuring transformations to the SPARK00 benchmarks and show that in ideal cases, speedups exceeding 20 are possible by regularizing memory reference streams in combination with structure splitting. Of course, the run-time introduces a considerable amount of overhead and is a constant component in our benchmarks. We will consider this overhead separately in Section 6.4.2 to allow a better comparison between the different data sets.

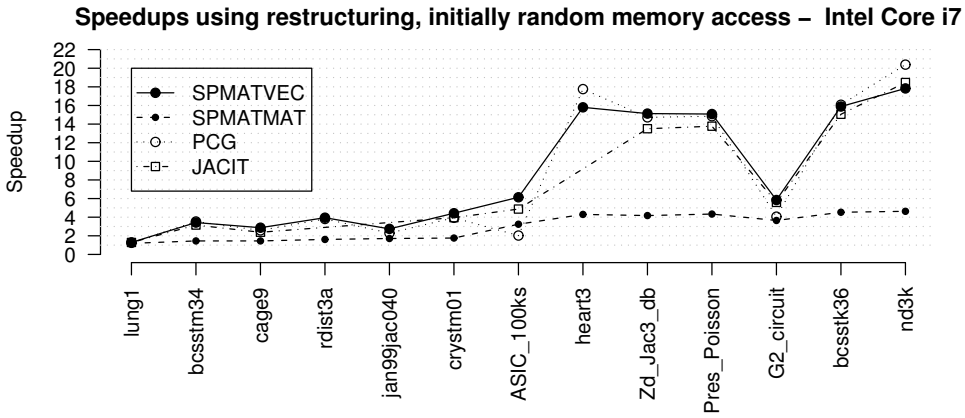
Figure 6.3(a) and 6.3(b) show the results of restructuring on the pointer-based SPARK00 benchmarks (except DSOLVE, which is treated separately), if the initial data layout causes random memory access, on the Intel Core 2 and Core i7, respectively. The data set size increases from left to right. As shown in previous work [114], optimizing data layout of smaller data sets is not expected to improve performance that much and this fact is reflected in the results. On both architectures, restructuring had no significant effect for data sets fitting into L1 cache. These sets have not been included in the figures. For sets fitting in the L2 and L3 cache levels, speedups of $1 - 6\times$ are observed. The Core i7 has an 8 $MiB$ L3 cache, whereas the Core 2 only has two cache levels. This explains the difference in behavior for the matrix *Sandia/ASIC_100ks*, which shows higher speedups for the Core 2 for most benchmarks. However, it turns out that the Core i7 runs almost $3\times$ faster when no optimizations are applied on SPMATVEC for this data set. Therefore, restructuring is certainly effective on this data set, but the greatest benefit is obtained when using data sets that do not fit in the caches.

An interesting case is DSOLVE, in which the lower triangle of the matrix is traversed row-wise, but the upper triangle is traversed column-wise. As the available data layouts of the matrices are row-wise sequential (CSR), column-wise sequential (CSC) or random (RND), none of these orders matches the traversal order used by DSOLVE. Figure 6.4(a) and 6.4(b) shows the results for DSOLVE using the different memory layouts on the Core 2 and Core i7, respectively. The matrices are ordered differently than in the other figures, as DSOLVE uses LU-factorized matrices as its input. These matrices have different sizes depending on the number of fill-ins generated during factorization. The matrices have been ordered from small to large (in the case of DSOLVE, this is the size after LU-factorization).

For the *lung1* data set, a decrease in performance is observed, but for the larger data sets, restructuring becomes beneficial again. Speedups of over $6\times$ are observed for the Core i7, using CSC (column-wise traversal would yield a sequential memory access pattern) as initial data layout. In principle, the RND (initial traversal yields a random memory reference sequence) data set could achieve much higher speedups, if after restructuring the best layout would be chosen. Currently, this is not the case for DSOLVE and we attribute this to the very simple permutation vector generation algorithm that we use (see Section 6.3.2). Generation of permutation vectors from

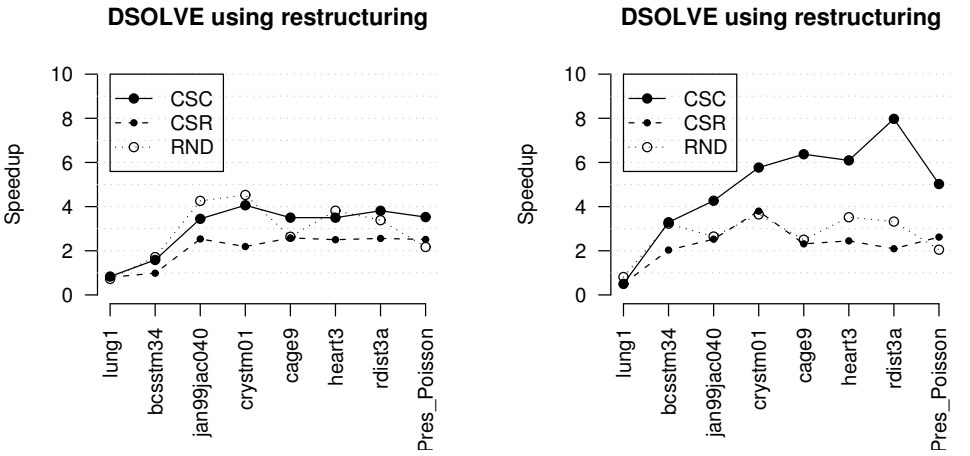



(a) Intel Core 2




(b) Intel Core i7

Figure 6.3: Speedups obtained using restructuring on the SPARK00 benchmarks. The initial data layout is random.




(a) Intel Core 2 (b) Intel Core i7

Figure 6.4: Speedups obtained using restructuring on DSOLVE for all different initial layouts. Input data sets are ordered by size (after LU-factorization).

traces will be improved in future versions of the framework.

### 6.4.2 Tracing- and Restructuring Overhead

Our framework uses tracing to generate a permutation vector that is used to rewrite the memory pool. Traces are kept for each field of a pool and one of these traces is used for restructuring. Currently, the trace to be used is specified as a compiler option, but this could potentially be extended to a system that autonomously selects the appropriate trace.

Tracing and the subsequent restructuring step have an impact on the performance. One cannot simply trace everything all the time as the system will run out of memory very quickly. In the benchmarks, we choose to only trace the first iteration of the execution of the kernel. In order to minimize the overhead of the tracing, the trace will only contain object identifiers, as described in Section 6.3.2. So for instance, if a linked list contains a floating point field and this list is summed using a list traversal, then both the pointer field and the floating point field are traced, there is an overhead of 2 trace entries per node visited. In our experiments, the structure operated on is 32 bytes and tracing the above mentioned traversal would add 16 bytes per node extra storage requirements when using 64-bit object identifiers. By using 32-bit objects identifiers, this would be reduced to 8 bytes. Subsequently, the memory pool is restructured using the information of the trace which relates to the field that contains the floating point values of the linked list nodes.

The overhead of the tracing and restructuring has been estimated by running a single iteration of each kernel with and without tracing and restructuring enabled,

| | *spmatvec* | | *spmatmat* | | *pcg* | | *jacit* | | *dsolve* | |
|---|---|---|---|---|---|---|---|---|---|---|
| Matrix | C2 | Ci7 | C2 | Ci7 | C2 | Ci7 | C2 | Ci7 | C2 | Ci7 |
| lung1 | 42.1 | 51.8 | 113.9 | 58.3 | 388.5 | 31.5 | 98.8 | 55.4 | N/A | N/A |
| bcsstm34 | 24.3 | 6.2 | 53.6 | 29.5 | 22.7 | 5.6 | 27.2 | 6.7 | 19.8 | 4.0 |
| cage9 | 21.0 | 8.1 | 44.3 | 26.1 | 22.1 | 8.1 | 28.6 | 10.3 | 2.9 | 2.0 |
| rdist3a | 17.9 | 5.6 | 39.5 | 21.1 | 17.7 | 5.2 | - | - | 3.2 | 2.1 |
| jan99jac040 | 16.0 | 8.0 | 16.3 | 15.3 | 17.8 | 8.2 | - | - | 1.1 | 1.3 |
| crystm01 | 8.3 | 4.9 | 17.1 | 17.0 | 9.1 | 4.9 | 10.8 | 5.8 | 2.2 | 1.8 |
| ASIC_100ks | 2.3 | 3.9 | 4.4 | 5.0 | 4.0 | 4.1 | 2.4 | 4.4 | - | - |
| heart3 | 2.4 | 1.7 | 4.6 | 4.8 | 2.4 | 1.5 | - | - | 3.1 | 2.2 |
| Zd_Jac3_db | 2.5 | 1.6 | 4.6 | 4.8 | 2.6 | 1.7 | 2.6 | 1.9 | - | - |
| Pres_Poisson | 2.6 | 1.7 | 4.7 | 5.0 | 2.6 | 1.7 | 2.7 | 2.0 | 3.8 | 3.0 |
| G2_circuit | 2.6 | 4.7 | 4.6 | 4.7 | 5.0 | 5.1 | 2.6 | 5.7 | - | - |
| bcsstk36 | 3.0 | 1.7 | 5.1 | 5.0 | 3.1 | 1.8 | 3.1 | 2.0 | - | - |
| nd3k | 3.5 | 1.9 | 5.4 | 5.2 | 3.5 | 1.9 | 3.6 | 2.1 | - | - |

Table 6.3: Number of iterations for the break-even points when tracing and restructuring is enabled, when using an initial random data layout. The matrices are ordered by increasing size. The lower part of the table contains the larger data sets, which do not fit in the caches. DSOLVE performs worse using *lung1* therefore a break-even point is not applicable. The missing entries for JACIT are due to zero elements on the diagonal. For DSOLVE the missing entries are due to matrices that take too long to factorize.

using a data layout causing random memory access. Figure 6.5 and 6.6 show the interpolated execution times of the benchmark PCG, both with and without restructuring for the Core 2 and Core i7 architectures. The initial data layout produces random memory access behavior of the application, which is eliminated after the first iteration when tracing and restructuring is used. After the first iteration, the application switches automatically to the non-traced version, which uses the restructured data. Four different matrices have been used which are representative in terms of performance characteristics (see Figure 6.3(a) and 6.3(b)). The break-even points for all matrices are included in Table 6.3.

The figures show that tracing does come with an additional cost, but for most (larger) data sets the break-even point is reached within only a few iterations. For instance, for all data sets shown in Figure 6.5, the break-even point is reached within 4 iterations, except for *cage9*, which is the smallest data set depicted. Interestingly, on the Core i7, the break-even point is reached even quicker, making restructuring more attractive for this architecture.

### 6.4.3 Run-time Stack Overhead

In order to quantify the overhead from the stack management that is needed if pool restructuring is desired, a few custom programs have been written. The interesting measurement in this case will be the overhead per function and per pointer.

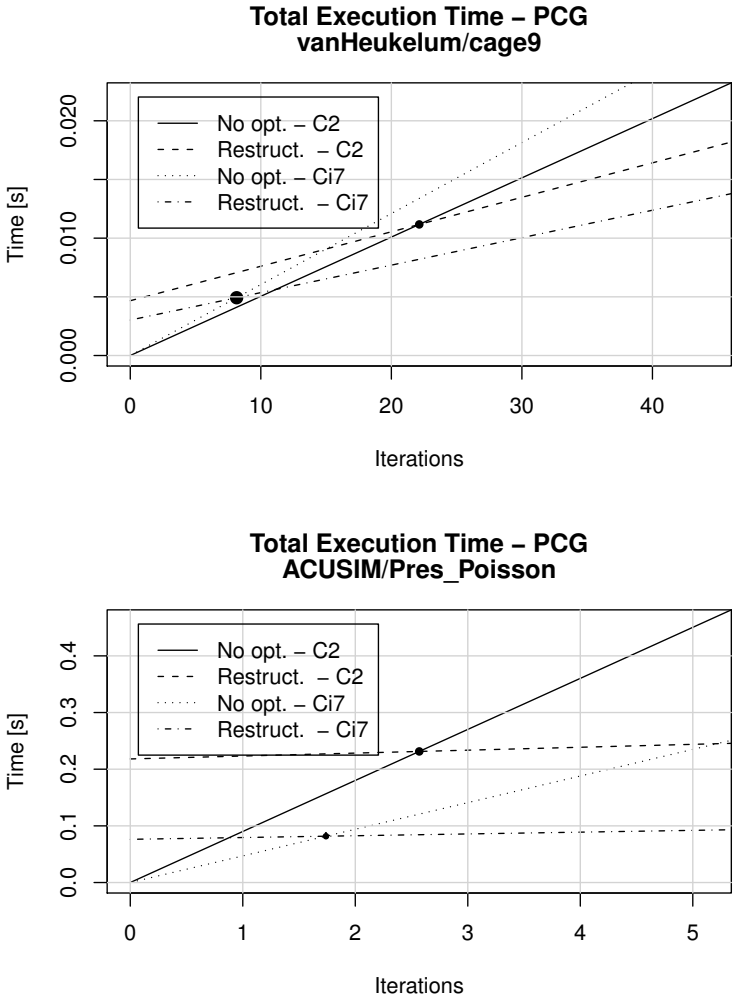



Figure 6.5: Execution times with and without restructuring. The break-even points are marked with a dot. The break-even points are reached earlier on the Core i7. Continued in Figure 6.6.

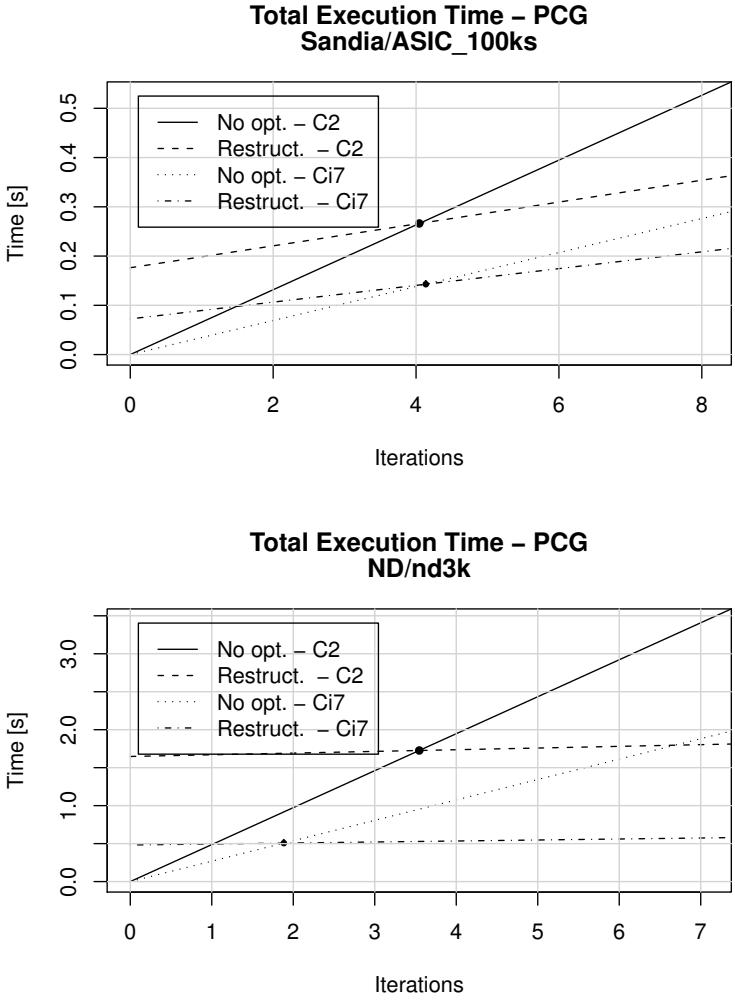



Figure 6.6: Execution times with and without restructuring. The break-even points are marked with a dot. The break-even point is reached earlier on the Core i7 for Sandia/ASIC_100ks. For ND/nd3k, the break-even point is reached slightly earlier on the Core 2.

In order to measure this overhead an experiment was carried out where a function is called that declares (and links together) a certain number of pointers that point into a pool. This was repeated for a multiple number of pointers and for both a version of the program built without the semi-managed stack and one version that was built with the semi-managed stack enabled. The function was executed a certain number (over a million) of times.

The following code demonstrates how this experiment was conducted:

```
listelem_t*
nextElem(list_t *list)
{
  if (list->current)
  list->current = list->current->next;

#pragma MAKE_POINTERS

  return list->current;
}
```

where the MAKE_POINTERS pragma was replaced by:

```
listelem_t *a0 = list->current;
listelem_t *a1 = a0;
listelem_t *a2 = a1;
listelem_t *a3 = a2;
...
listelem_t *aN-1 = aN;
```

The execution time for the loop calling the *nextElem* function was measured and the difference between the managed version and the unmanaged version should thus represent the overhead introduced for that number of pointers in the given number of calls to the function.

Figure 6.7 shows the execution time on a 2.5GHz *Intel Core 2 Duo*, of 4 million calls to the function above in several runs with different numbers of pointers declared and used in one function. The data evaluates to a base cost of 5 cycles per pointer being linked, for the pointer tracking alternative the cost is around 27 cycles per pointer being registered and linked. This gives the penalty of explicit pointer tracking to 22 cycles per pointer being tracked. This overhead is obviously quite substantial, but the compilation chain described in this chapter employed a simple optimization in order to minimize the overhead.

The optimization used was based on disabling the pointer tracking when not needed, for example in descendant functions from the one that calls the restructuring run-time (since the stack on the descendants will be dead anyhow, when the restructuring function is invoked).

Since the shadow stack and stack map strategies were never implemented, the test case was modified to include simulation code in order to estimate any overhead, though it was expected that such overhead would be minimal.

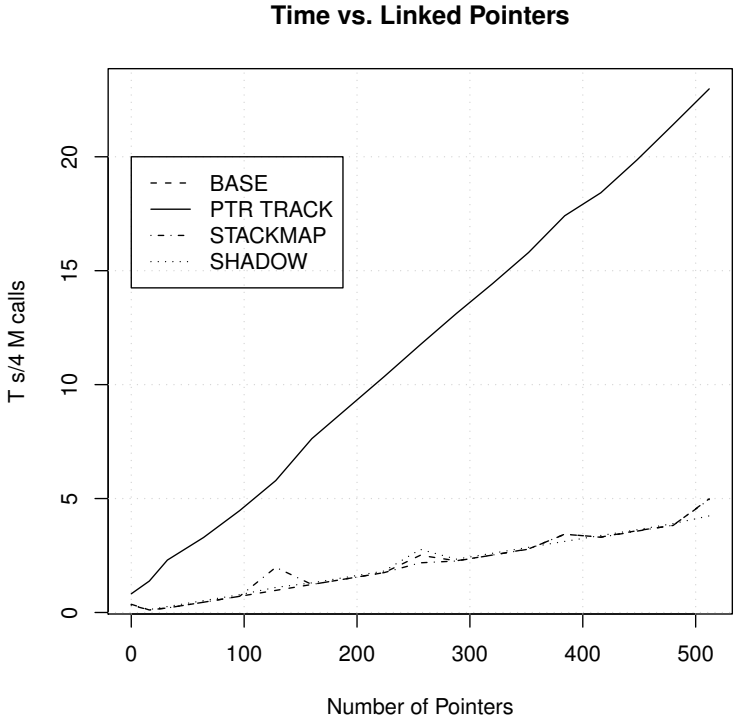



Figure 6.7: Execution time of a function with different stack management approaches

By pooling all the pointers associated with a pool in a function into a single per function data structure, it is possible to eliminate all per pointer overhead associated with registering each pointer. In this case, only the address of the record containing all the pointers would need to be registered. This has its own problems, as it prevents certain optimizations to be run on the code such as the elimination of unused pointers (though the pointer tracking suffers from the same issue).

The stack map approach offers none of the run-time overhead (except during the stack walks when program counter entries on the stack are translated into function ids), but does on the other hand require modifications in the compiler's backend.

### 6.4.4 Address Calculations

The address calculation expressions used are an improved variant of those introduced by Curial et.al. [35]. These improvements have been verified experimentally by running two versions of the pointer-based applications from the SPARK00 benchmark suite [114], one with the new optimized address calculation expressions enabled, and

| *Bench Name* | *Address Calc Improvements* |
|---|---|
| *DSOLVE* | 4.87 % |
| *JACIT* | 4.59 % |
| *PCG* | 1.99 % |
| *SPMATMAT* | 3.81 % (6.22%/4.16%/1.05%) |
| *SPMATVEC* | -6.11 % |

Table 6.4: Average performance gains (in percent) for pool allocation and the improved field offset equations. Note that SPMATVEC has a negative improvement due to instruction cache conflicts. For SPMATMAT, different figures are given in parentheses for 1, 7 and 30 columns in the right hand matrix, respectively.

one version with only the general addressing equations used by MPADS enabled. It should be noted that the implementation described here is not using the same compiler framework as MPADS which is based on XLC. Thus a direct comparison between Curial's work and the compiler chain introduced here has not been carried out.

The matrix input files are sparse and inserted in row-wise order, leading to a regular access pattern when traversing the data structure. In Figure 6.8, the matrices are ordered by size. For the *SPMATMAT* benchmark the same matrices are used three times each: one pass using one column of the right hand side matrix, the second pass using seven columns and the third pass using 30 columns. Note that the matrix multiplication in *SPMATMAT* is multiplying a *sparse* matrix with a *dense* matrix. The result of this multiplication is a dense matrix.

*SPARK00* was compiled with *LLVM GCC* in order to generate *LLVM* bit code. The bit code was then passed through the *LLVM* bitcode linker, after which it was transformed by the pool allocation (Lattner [73]) and structure splitting optimization passes.

When running the experiments, it was expected that the new field offset equations will in principle never be less efficient than the generic ones, excluding effects on instruction caches and any reordering that the compiler may do or forgets to do due to the changed instruction stream.

Table 6.4 gives the average improvements of the addressing optimizations. Note that in Table 6.4, the SPMATVEC benchmark actually lost in performance, this was due to instruction cache conflicts in the new code, and thus not related to the address calculation expressions themselves. Figure 6.8 show the general behavior of the benchmarks where the relative performance improvements is greater for smaller data set sizes, this is because the new instruction mixture actually plays a greater part in those cases. For the larger data sets, the performance is more bounded by the memory latency and thus the instruction mixture has less influence in total.

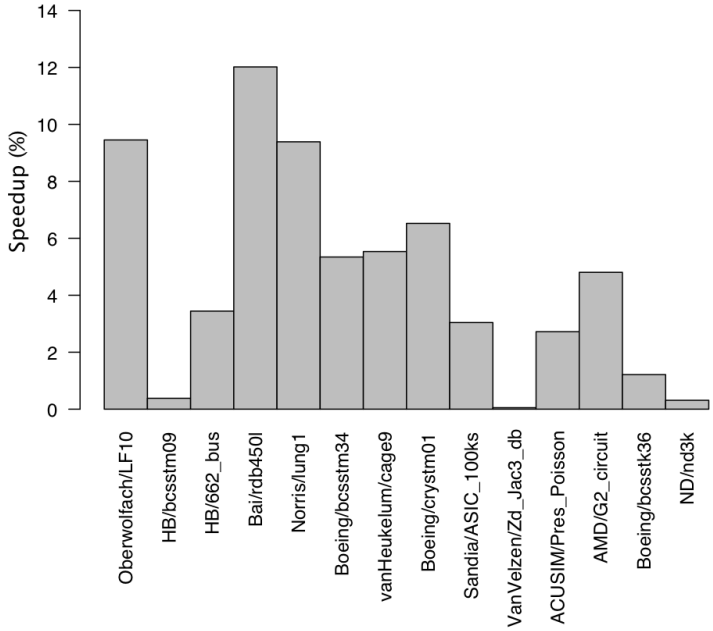



Figure 6.8: Typical improvements (in this case for the *JACIT* benchmark) ordered by increasing matrix size, from left to right.

## 6.5 Summary

In this chapter, we presented and evaluated our restructuring compiler transformation chain for pointer-linked data structures in type-unsafe languages. Our transformation chain relies on run-time restructuring using run-time trace information. We have shown that the potential gains of restructuring access to pointer-based data structures can be substantial.

Curial et al. mention that relying on traces for analysis is not acceptable for commercial compilers [35]. For static analysis, this may often be true. For dynamic analysis, which does not need the intervention of the person compiling the application, relying on tracing is not necessarily undesirable and we have shown that the overhead incurred by the tracing and restructuring of pointer-linked data structures is usually compensated for within a reasonable amount of time, if data structures are used repetitively.

Our restructuring framework opens up more optimization opportunities that we have not explored yet. For example, after data restructuring extra information on

the data layout is available which could be exploited to apply techniques such as vectorization on code using pointer-linked data structures. This is a subject of future research.

Data structures that are stored on the heap contain object identifiers instead of full pointers. This makes the representation position-independent, which provides new means to distribute data structures over disjoint memory spaces. Translation to full pointers would then be dependent on the memory pool location and the architecture. This position independence using object identifiers has been mentioned before by Lattner and Adve in the context of pointer compression [76]. However, with our pool restructuring, a more detailed segmentation of the pools can be made and restructuring could be extended to a distributed pool restructuring framework.

The implementation described in this chapter uses some run-time support functions to remap access to the proper locations for split pools. The use of object identifiers implies a translation step upon each load and store to the heap. These run-time functions are efficiently inlined by the LLVM compiler and have a negligible effect when applications are bounded by the memory system. The run-time support could in principle be implemented in hardware and this would reduce the run-time overhead considerably. We envision an implementation in which pools and their layout are exposed to the processor, such that address calculations can be performed transparently. Memory pools could then be treated similarly to virtual memory in which the processors also takes care of address calculations.

We believe the restructuring transformations for pointer-linked data restructures that have been proposed in this chapter do not only enable data layout remapping, but also provide the basis for new techniques to enable parallelizing transformations on such data structures.

# CHAPTER 7

# Enabling Array Optimizations on Code Using Pointer-Linked Data

For decades, most code was written under the assumption that there would be only one processor available. This, together with other factors, led to particular concepts and techniques used to write programs. One of these concepts, which originates from the way main memory is accessed (as one large vector of bytes), is the concept of *pointer* variables that point to a location in memory. While technically a pointer is just an address, programming languages add some extra information about the type of the data that is stored at the location pointed to. For example in C, this might be a 64-bit integer, or a structured type.

Another commonly used concept is the array. This is just a linear sequence of elements of the same type in memory. Especially arrays of atomic (i.e. non-structured) values have been used extensively and many analyses and optimization methods target code using arrays. Key to their susceptibility to optimizations is their predictability. At compile time, it is known that every element has the same type. The computation of addresses of elements within arrays is straightforward, namely $\mathit{offset} \times \mathit{sizeof}(\mathit{Type})$. Iteration over such arrays is often done using simple, counted loops.

So what is the point here? While successful optimizations for array-based code have been implemented (even parallelizing transformations), such optimizations could never be directly applied to code using pointer-linked data structures. This has several causes. In pointer-based code, the address right after the current data element might be of a different type. Traversing a pointer-linked data structure potentially yields a scattered memory reference pattern, while access to arrays stays within the array bounds. The loops used to traverse the data structure are usually not counted loops. Pointer-chasing loops with data-dependent loop conditions are characteristic for such traversals.

Ideally, optimizations that are designed for array-based code should also work on code containing pointer-linked structures. In this chapter, a combined compile/run-time approach is proposed that enables an array-based view on pointer-linked data structures, thereby paving the way to the application of more advanced compilation techniques, such as used in the case study in Chapter 8. Compile-time analysis alone is certainly not going to be the solution to the problem of eliminating pointers and other forms of irregularities arising from the use of pointer-linked structures. For example, in order to detect if data-dependent loop conditions did not change between two function calls, run-time support is required. Three key properties of pointer-linked code make the optimization of such code different from optimizing array-based code: *data is stored as linked elements of structured data, loops are not simple, counted loops and memory access patterns are unpredictable*. In this chapter, transformations are presented that generate code and a data layout whose properties are more like that of array-based code.

The analysis algorithms, transformations and run-time components that are presented in Section 7.1 have been implemented using the LLVM compiler framework. The transformations described in this chapter explicitly target to enable traditional array and control flow optimizations on code using pointer-linked data structures. These transformations depend on many properties of Data Structure Analysis (DSA) and structure splitting, which have been discussed in more detail in Chapter 5. In Section 7.2, we evaluate the results of our transformation chain. The overhead of the run-time mechanisms are quantified, and we apply the transformation chain on the SPARK00 benchmarks (sparse matrix computations using pointer linked data structures). Section 7.3 summarizes this chapter.

## 7.1 Control Flow Optimization of Pointer-Based Code

Pointer-based code often uses control flow structures that are not easy to analyze. Loop conditions might be data dependent and long dependency chains are common. At first, any comparison with counted loops seems far fetched, but if we take a closer look at this problem, they are not that different in many cases. The key to this similarity in behavior is *non-modified* data. In many cases it cannot be proved that control flow behaves in a particular way at compile-time. However, we do know that for any function, its loops will execute exactly the same way as during the previous call, if all data that the control flow depends on is not modified. In this section, we describe our heap-analysis and loop transformation that changes data dependent well-formed loop structures into simple, counted loops.

### 7.1.1 Data Dependencies in Pointer-Based Code

In pointer-based code, data dependencies are not as easily detected as in array-based code. For example, if a function is called with a particular pointer argument and later on it is called a second time, with exactly the same pointer as its argument, this does not mean that the function will execute the same instruction sequence. This is caused

by the underlying data structure. While the actual function argument might be the same, the data it points to, together with all the other data that is being pointed to (when using recursive data structures), might be completely different.

Therefore, in order to determine whether control flow for loops will be the same on successive calls of a function, two types of dependencies must be identified: *variable dependencies* and *heap data dependencies*. The analysis presented in this section takes both these types of dependencies into account. Pointers to stack data are not supported as such data is not pool-allocated. In order to pool allocate stack data, stack allocations have to be rewritten to heap allocations. This is a non-trivial task, as such allocations should be automatically freed when the function returns, even when exceptions occur or functions like *setjmp* and *longjmp* are used. Therefore, this feature has not been implemented so far.

### 7.1.2 Data Dependence Analysis for Loop Conditions

We have implemented a data dependence analysis for loop conditions in pointer-based code using the LLVM compiler framework [74]. This analysis has as its input a whole program and computes for each function both the *variable dependencies* and the *heap data dependencies* for conditions of nested loops. Algorithm 1 and 2 show the data dependence analysis for loop conditions.

#### Algorithm for the Identification of Data Dependencies

The algorithm calls $findLoopDepRoots$ for each function. $findLoopDepRoots$ starts by taking the terminator instruction of each exiting block of a loop and it inserts the condition variable (which determines control flow) in the $traceBackSet$ set. This set contains all variables that have to be traced back to their root, which is either a local (stack) variable, a function argument or a global variable. The algorithm proceeds by picking a variable from $traceBackSet$ (in LLVM, this is an llvm::Value object):

- If this value is an argument or a global variable, it is *root* of the data dependencies. Otherwise, it is a variable that is defined within this function, and therefore it is a *derived* value. Arguments and global variables are added to the root set, and in case it is a pointer value we check if it is pointing to pool-allocated data. If it is not, the function is unsafe and we should not proceed. Otherwise, we add the pool to the set of pools ($DSDeps$).

- Else, if this value is the result of a load instruction (it is of the type llvm::LoadInst), we need to check whether this loads data from the local stack or from a memory pool that is split. Other cases cannot be analyzed and are classified as unsafe. As the control flow depends on the value of this load, all values that are stored to this location within the current function must be identified and added to $traceBackSet$. This is done by determining all possible aliases of the pointer operand of the load instruction using the DSA-based alias analysis available in the source distribution of DSA. Subsequently, all values

```
function findLoopDepRoots( function )
begin
    // DSA-based alias analysis
    DSAA = getDSAAForFunction(function)
    traceBackSet = ∅
    loopDepRoots = ∅
    DSDeps = ∅
    // Initialize set of variable dependencies with loop conditions
    foreach loop in function do
        foreach exitingBlock in loop do
            termInst = exitingBlock.getTerminator()
            c = termInst.getCondition()
            traceBackSet = traceBackSet ∪ {c}
        end
    end

    // Trace back values until a function argument or global is found
    variableDone = ∅
    while traceBackSet != ∅ do
        value = traceBackSet.get()
        traceBackSet.remove(value)
        variableDone = variableDone ∪ value
        if isArgument(value) or isGlobal(value) then
            loopDepRoots = loopDepRoots ∪ value
            if isPointer(value) then
                if getPoolInfoForValue(value) then
                    DSDeps = DSDeps ∪ getPoolForValue(value)
                else
                    mark function UnSafe
                    return
                end
            end
        else if isLoadInst(value) then
            // See Algorithm 2
            analyzeLoad(value, traceBackSet,
                    accessFieldsForType, DSAA )
        else if isInstruction(value) then
            inst = cast<Instruction>(value)
            foreach operand of inst do
                if operand ∉ variableDone then
                    traceBackSet = traceBackSet ∪ operand
                end
            end
        end
    end

    findModGlobals(function)

    // Check if data on which control flow depends is not
    // modified in the same function and its callees
    foreach pool ∈ DSDeps do
        if DSNode for this pool has MOD flag set then
            mark function UnSafe
            return
        end
    end
end
```

**Algorithm 1:** Algorithm to determine root data dependencies of loop conditions. This detects function arguments, global variables and the heap-allocated data that might be pointed to.

```
function analyzeLoad(loadInst, traceBackSet,
        accessFieldsForType, DSAA )
begin
    if accessesSplitPool(loadInst) then
        field = getAccessedField(loadInst)
        PI = getPoolInfo(loadInst)
        accessedFieldsForType[PI.getType()].insert(field)
    else if not isLocalStackLoad(loadInst) then
        return Unsafe;
    end

    aliasSet = DSAA.getAliasSetForPointer(
                    loadInst.getPointerOperand())
    // If some potential alias of this pointer is modified
    if aliasSet.isMod() then
        foreach store to any pointer target in aliasSet
                within this function do
            // Add the value that is stored to the set
            // of variables that must be traced back
            traceBackSet.insert(store.getValueOperand())
        end
    end
end
```

**Algorithm 2:** Analysis of load instruction during identification of dependencies. Identifies all pointers that might be aliased to the load target. If there is a store instruction storing to any of these pointers, the value of that store is added to the set of dependencies that must be traced back to their roots.

stored within the current function to any potential alias of the pointer variable are added to *traceBackSet*.

- Else, if this value is the result of another instruction (it is of the type llvm::Instruction), add the operands of this instruction to *traceBackSet*.

*findModGlobals* determines which global variables may be modified by the function. It considers all store instructions within the strongly connected component of the control flow graph that the function belongs to and if the target stored to is flagged as global in the DS graph, the pointer operand of the store instruction is added to the set of modified global variables. If not all call targets can be detected (unresolvable indirect calls), then all variables are marked as MOD.

The function to be optimized (and its callees) should not modify any data on which the loop conditions depend. This information is available statically from the Bottom-Up DSA, which describes how data is accessed from within a function *and* its callees. Unfortunately, DSA is currently field-insensitive [72], as opposed to our run-time MOD flag tracing. Lattner has stated that this would be easy to add to their implementation, but this has not been done, yet [72]. Implementing field-sensitivity would allow for modification of fields in structures that do not affect control flow.

```
struct CFOptRecord
{
  struct {uint64_t poolid1; uint64_t poolid2; ...};
  struct {uint64_t modflagsmask1; uint64_t modflagsmask2; ...};
  struct {Type1 arg1; Type2 arg2; ...};
  struct {TypeG1 global1; TypeG2 global2; ...};
  struct {uint64_t *traceVector1, uint64_t *tracevector2, ...};
  struct {uint8_t hasTrace};
};
```

Figure 7.1: Run-time control flow optimization record. This stores information about the last call to the associated function, such that immutability of control flow of the loops within the function can be determined.

#### Instrumentation for Run-time MOD Flag Tracking

Knowing all arguments, globals and pools on which the loop conditions depend, the next step is to instrument stores to fields in pools on which these conditions depend, such that a flag is set whenever a field has been modified. This is analogous to the MOD flags in DS Nodes as used in DSA. The difference is that our mechanism provides run-time MOD flags at the field level, while DSA provides compile-time MOD information at the whole-structure level. All store instructions are checked if their storage types need MOD flag tracking, and if so, code is inserted that sets the proper MOD flag for the field accessed in the pool descriptor (a run-time structure storing pool properties).

### 7.1.3 Loop Rewriting

At this point, we know for each function which globals, arguments and when dealing with pointers, which fields in pools the loop conditions depend on. Thus, if none of this dependencies change, control flow will be exactly the same as during the previous call of the function. As we only deal with natural loops (LLVM's LoopInfo analysis gives results on natural loops, which are well-formed, that is, they only have one unique entry point), these loop structures can be replaced by simple, counted loops. Loops are guaranteed to be in a normal-form, that is, they only have a single back edge and a single exit basic block (the block executed directly after exiting the loop). Any predecessor for such an exit block is always within the loop.

In order to obtain counted loops, the original function (which is executed when iteration counts are not yet known) must record the loop bounds for each execution of a loop, that is whenever the loop pre-header is entered. Figure 7.1 depicts the run-time meta data maintained for each function that can be transformed. At run-time, the record is used to keep track of the arguments and memory pools passed in at the last call. The *modflagsmasks* are masks which have the bits set for the fields on which the loop conditions are dependent (currently this is a 64-bit vector, thus a

maximum of 64 fields per pool is supported). The values of the function arguments at the last call are also stored in this record and the same holds for global values. The *trace vectors* are pointers to integer arrays, which contain the loop bounds each time a loop is entered. *hasTrace* is a flag which is set after the trace vectors have been initialized.

```
function insertLoopBoundTracing( function )
begin
    foreach loop that is traceable do
        entry = getEntryBasicBlock(function)
        entry.insert(loopBoundCounter)
        cfoptrecord.traceVector = allocateTraceVector()
        entry.insert( "loopBoundCounter = 0" )
        entry.insert( "loopBoundIdx = 0" ) // Offset trace vector
        latch = loop.getBackEdge()
        latch.insert("loopBoundCounter++")

        exitBlock = loop.getExitBlock()
        exitBlock.insert(
            "traceVector[loopBoundIdx] = loopBoundCounter" )
        exitBlock.insert( "loopBoundCounter = 0" )
        exitBlock.insert( "loopBoundIdx++" )
    end
end
```

**Algorithm 3:** Insertion of run-time loop bound tracing in original function. These traces are used as loop boundaries in the optimized function.

Algorithm 3 shows the pseudo-code that generates the necessary code for computing the run-time loop bound information. It is most easily illustrated by an example. Figure 7.2 shows a simple example traversal of a data structure in C. In Figure 7.2a, a simple pointer traversal is shown. In real life "one ought to do something useful during the traversal", but let us keep things clean for the sake of clarity. In Figure 7.2b, the same code is shown with the run-time tracing inserted. The *loopBoundCounter* variables are incremented at every back edge and after exiting the loop, the counter is written to the trace vector. At the end of the function, the *hasTrace* flag is set in the optimization record. If upon the next function call the right conditions are met (see the next section) the code in Figure 7.2c is executed. The pointers are still there, and they will be dealt with later.

### 7.1.4 Function Dispatch Mechanism

At this point, there are two functions at our disposal. The first is the original function, which now performs tracing to obtain loop counters, the second is the newly generated function, which uses counted loops. The second function should only be called instead of the first one if safety conditions are met. These conditions are checked at run-time. We will describe the function dispatch mechanism in this section.

Figure 7.3 depicts the function selection process. First, it is checked whether the optimization record has its *hasTrace* flag set. If it is not set, the function which generates the trace vectors with iteration count limits must be called. In the case that there is a trace available, we must check whether any data on which loop conditions

(a)

```
while (p) {
  p2 = p;
  while (p2) {
    p2 = p2->down;
  }
  p = p->right;
}
```

(c)

```
loopBoundIdx1 = 0;
loopBoundIdx2 = 0;
for (i = 0; i <
traceVector1[loopBoundIdx1];
i++ ) {
  p2 = p;
  for (j = 0; j <
traceVector2[loopBoundIdx2];
j++ ) {
    p2 = p2->down;
  }
  loopBoundIdx2++; p =
p->right;
}
loopBoundIdx1++;
```

(b)

```
loopBoundCounter1 = 0;
loopBoundCounter2 = 0;
loopBoundIdx1 = 0;
loopBoundIdx2 = 0;
traceVector1 =
   cfOptRecord.traceVector1;
traceVector2 =
   cfOptRecord.traceVector2;

while (p) {
   p2 = p;
   while (p2) {
     p2 = p2->down;
     loopBoundCounter2++;
   }
   ...
   ...
   traceVector2[loopBoundIdx2]
   = loopBoundCounter2;
   loopBoundCounter2 = 0;
   loopBoundIdx2++;

   p = p->right;

   loopBoundCounter1++;
}
traceVector1[loopBoundIdx1]
  = loopBoundCounter1;
loopBoundCounter1 = 0;
loopBoundIdx1++;
cfOptRecord.hasTrace = 1;
```

Figure 7.2: Example of the code transformation to counted loop structures. (a) shows a nested pointer traversal. In (b), each loop is instrumented to keep track of the iteration counts which are stored to the trace vectors. Whenever it is determined that the control flow of the loops did not change, (c) is executed using the loop counts from the trace vectors in the counted loops.

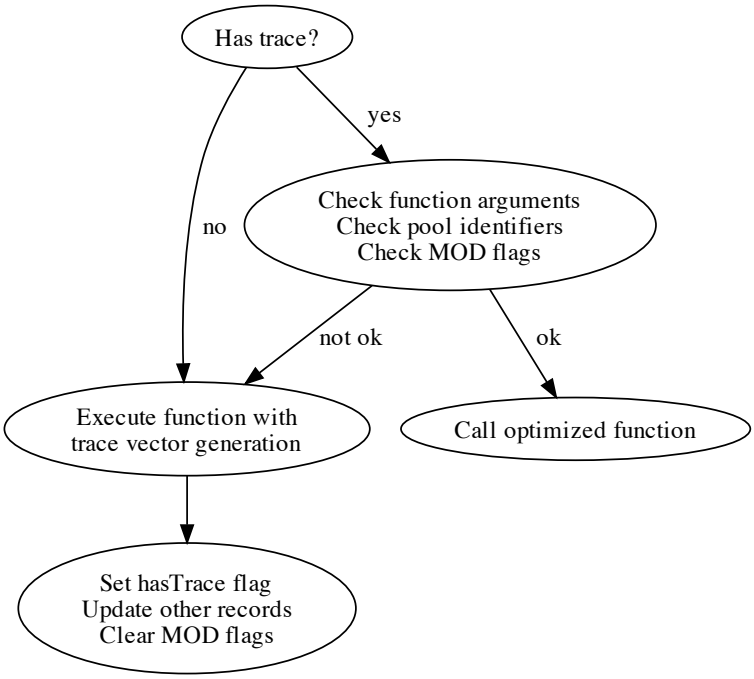



Figure 7.3: The function dispatch mechanism checks if a trace exists, if the function arguments and pool-allocated data that the loop conditions depend on did not change and selects the proper function to call accordingly.

depend has been changed. This is done by checking whether the function is called with the same actual arguments (the value), and by checking if the data they point to have changed. In addition, the current *pool identifier* is checked against the previous, to ensure the same memory pool is used. Then, the run-time MOD flags (which are stored in the pool descriptor structure) are checked against the mask stored in the optimization record. If none of the fields have changed, the optimized version can be called.

In the case that any of these conditions is not met, the function that generates a new trace is used. This is the same situation as when the *hasTrace* flag is not set. After generation of the trace vectors, the *hasTrace* flag is set for this specific function (in the optimization record). All other records are checked, as when the run-time MOD flags in the pool descriptor are cleared, other records might become inconsistent. Therefore, it is checked if any set MOD flag would invalidate the trace vectors of other optimized functions, and if so, the *hasTrace* flag for that specific function is cleared.

### 7.1.5 Converting Pointers to an Array-Based Representation

For pool allocated type-homogeneous data structures, pointers can be represented by an object index into the pool. Lattner and Adve mention this possibility in their pointer compression implementation [76] and we have used this representation in Chapter 6 in the implementation of automatic restructuring of linked data structures. In the latter framework, all pointers that are on the stack and in LLVM virtual registers are represented as normal pointers and all pointers that are stored to pools are stored as object identifiers. We follow the same strategy, but for the generated code with counted loops, all pointers are stored as object identifiers and are only converted back to pointers if they escape the current function as a function argument. All incoming pointer arguments are immediately converted to object identifiers.

As all pools have been split already, data is already grouped by field, and the pool descriptor contains the pointer to the start of data of each field. Figure 7.4 shows how the code with the counted loops from Figure 7.2 is transformed. For each split structure, the pool meta data is passed into the function as a pool descriptor. Among other information, this pool descriptor contains the base pointer for each field of the associated structured type. At the start of the function, all the array pointers of the fields from pools that are used in the function are loaded. Any computation of an address within a pool that is load from or stored to can now be computed by indexing the array using object identifiers. This transformation in itself does not directly affect performance, but it results in a representation on which existing analyses can be applied.

### 7.1.6 Controlling Memory Access Patterns

Memory access patterns can be very unpredictable when running pointer-based code. This unpredictability does not magically disappear if the code is rewritten in the array-based form as explained above, e.g.

```
IndexType *ArrayBase_Right = Pool->field_ptr[fieldNoOfRight];
IndexType *ArrayBase_Down  = Pool->field_ptr[fieldNoOfDown];
loopBoundIdx1 = 0;
loopBoundIdx2 = 0;

for (i = 0; i < traceVector1[loopBoundIdx1]; i++ ) {
  p2_oid = p_oid;
  for (j = 0; j < traceVector2[loopBoundIdx2]; j++ ) {
    p2_oid = ArrayBase_Down[p2_oid];
  }
  loopBoundIdx2++;
  p_oid = ArrayBase_Right[p_oid];
}
loopBoundIdx1++;
```

Figure 7.4: Pointer to array conversion converts all pointers to fields within a split pool into an offset with respect to the base pointer of the array used for that field. This figure shows the result of converting the code from Figure 7.2c to an array-based representation.

```
p_oid = ArrayBase_Right[p_oid]
```

is basically still a pointer chasing statement. The fact that everything is expressed using arrays and indices makes things easier, as data access is at least confined to the region allocated for the array, and within this region, only data of the same type can be expected.

In this section, we describe three different forms of memory access, between which an application can switch (under some conditions) using our framework: *unprefetchable indirect access*, *prefetchable indirect access* and *direct access*. Figure 7.5 shows the basic examples of these three different forms. Unprefetchable indirect access is memory access that cannot be initiated before a previous memory access has been completed. Such a type of memory access often suffers from high latency (except

```
k = x;
for (i=0; i<n; i++) {
  k = A[k];
  ... = B[k];
}
```
(a)

```
for (i=0; i<n; i++) {
  k = A[i];
  ... = B[k];
}
```
(b)

```
for (i=0; i<n; i++) {
  ... = B[i];
}
```
(c)

Figure 7.5: Three different forms of memory access in loops. (a) shows unprefetchable indirect access, (b) prefetchable indirect access and (c) shows a loop using direct access.

when hardware prefetching is successful). Prefetchable indirect access is a memory access that can be initiated independent from other memory accesses. In the example in Figure 7.5b, $A[i]$ can be fetched independently for each iteration, and latency is not a problem anymore. The best option of course is direct access. No memory access to find an array index is needed, and this is of course the most efficient. Additionally, direct access often results in a sequential access pattern, which is memory-subsystem friendly, and enables other optimizations such as vectorization.

The worst case is that memory access is unpredictable, and that indices are not prefetchable. In our example, let us assume that the index array $A$ is not being written to. It is only used to fetch the next $k$. Let $k_i$ be the value of $k$ at iteration $i$. During execution, these values can be traced and stored in a temporary array $X$. After executing the unprefetchable version, $k_i == X[i]$ holds. So upon the next execution, if the array $A$ has not been changed (no run-time MOD flag has been set for the field that is associated with in our framework), $X$ can be used instead. Substituting `k = X[i]` for `k = A[k]` yields prefetchable, indirect access to $B$, the same code as shown in Figure 7.5b.

The next step is harder, to move from indirect access to direct access. Eliminating indirect access is possible, if $A[i] == i$. In that case, $i$ can be substituted for $k$ in our example. However, it would be sheer luck if this is the case in an application. In general, if a sequential access pattern is needed, the data should be reordered. In Chapter 6, we have seen how restructuring of linked data structures has been implemented, which is based on pool-allocated and split data structures (remember that all the arrays we are talking about here have their origin in the split structures). Therefore, the data can be reordered in memory by applying a permutation to the split memory pool. In the case that the array $X$ mentioned above is an injective mapping, the memory pool can be reordered such that $X[i] == i$ holds (referring data is correctly updated by the restructuring algorithm).

## 7.2 Experiments

The optimizations described in the previous sections are not meant to give great performance improvements by themselves. Rather, they try to transform the code into a representation that is recognized by other existing optimizations. All the transformations described in the previous section have been implemented within the LLVM framework, except for the code transformations described in Section 7.1.6.

In this section, we will focus on the effect that our transformations have on performance, as our run-time system does have some overhead. First, we will show the performance impact of our function dispatch mechanism and the tracing of the loop bound trace vectors. Our transformation chain as a whole will be evaluated on the pointer benchmarks from SPARK00 (see Chapter 3). The benchmarks used are: SPMATVEC (sparse matrix vector multiplication), SPMATMAT (sparse matrix times dense matrix), DSOLVE (direct solver), JACIT (Jacobi iteration) and PCG (preconditioned conjugate gradient). The experiments are run on an Intel Core 2 Duo $2.33GHz$ with $4MiB$ of cache and $2GiB$ of main memory, running Mac OS X 10.6.2.

### 7.2.1 Overhead

At run-time, there are two mechanisms that incur overhead: the function dispatch mechanism that selects which function must be called, and the generation of trace vectors. In this section, we will first evaluate the overhead of the dispatch mechanism using a micro-benchmark specifically written for this purpose. Next, the overhead of the generation of trace vectors is estimated using the SPARK00 benchmark set.

#### Dispatch Mechanism

Our framework checks preconditions to see whether data on which loop conditions depend has changed. At each call site of a function that has an optimized implementation, all function arguments have to be checked against the values that the function was previously called with. In addition, run-time MOD flags must be checked for all pools on which loop conditions depend. After a function has generated its trace vectors, all MOD flags involved must be cleared.

We estimate the performance penalty caused by this dispatch mechanism by generating a C function for which the number of non-pool arguments and arguments with an associated pool can be specified. Figure 7.6 shows an example of the code that is used for two pool-allocated arguments and one normal function argument. This code is generated by a script, and any combination of non-pool variables and pool allocated objects can be created. The code calls the function *func* repeatedly. The first call will be to the split version of the function, the subsequent calls will be to the version with counted loops. Each time the function is called, the dispatch code is executed to determine the right call target. The MOD flags are cleared after executing the function that generates the trace vectors.

In order to make the function *func* depend on all variables the condition of the while loop includes a reference to all function arguments and each data element of the objects passed in. At run-time, this loop is never executed, as the if-statement will return immediately. However, due to the flow-insensitivity of the analyses, the if-statement will not prevent the generation of the dispatch mechanism.

In this experiment, the impact for non-pool variables and pool allocated objects is evaluated separately. For both types of variables, we measure the average overhead per function argument for up to a total of 31 function arguments. For the experiments, we fix the number of pool arguments to one and vary the number of normal arguments from 1 to 31, or we fix the number of non-pool arguments to 1 and vary the number of pool arguments from 1 to 31.

Figure 7.7 shows the results of the overhead per function argument (for pool allocated and normal arguments) per call. For the first three pool arguments, the cost per argument is decreasing after which the cost per argument increases until at around 17 pool arguments the cost per argument stabilizes. In this (worst) case, the average cost per pool-allocated argument is around $6ns$. This is around 15 times slower than a normal function call. While this is a relatively large performance penalty, it is still pretty small in the absolute sense and in the next section, it is shown that for data-intensive applications this extra overhead is negligible. For non-pool allocated

```
typedef struct _Object {
  uint64_t val;
  uint64_t val2;
} Object;

void func( Object *obj0, Object *obj1,
      uint64_t var0 )
{
  if( var0 == 0 ) return;
  while( obj0->val && obj0->val2 &&
      obj1->val && obj1->val2 && var0  ) {
    printf( "Hello world\n" );
  }
}

int main()
{
  unsigned i;
  unsigned lim = 100000000;
  uint64_t var0 = 0;
  Object *obj0 = (Object *)malloc(sizeof(Object));
  obj0->val2 = 0;
  obj0->val = 0;
  Object *obj1 = (Object *)malloc(sizeof(Object));
  obj1->val2 = 1;
  obj1->val = 1;
  for( i = 0; i < lim; i++ ) {
    func( obj0, obj1, var0 );
  }
  return 0;
}
```

Figure 7.6: Example code for the evaluation of the overhead of the dispatch mechanism. This example shows one non-pool variable and two pool allocated objects.

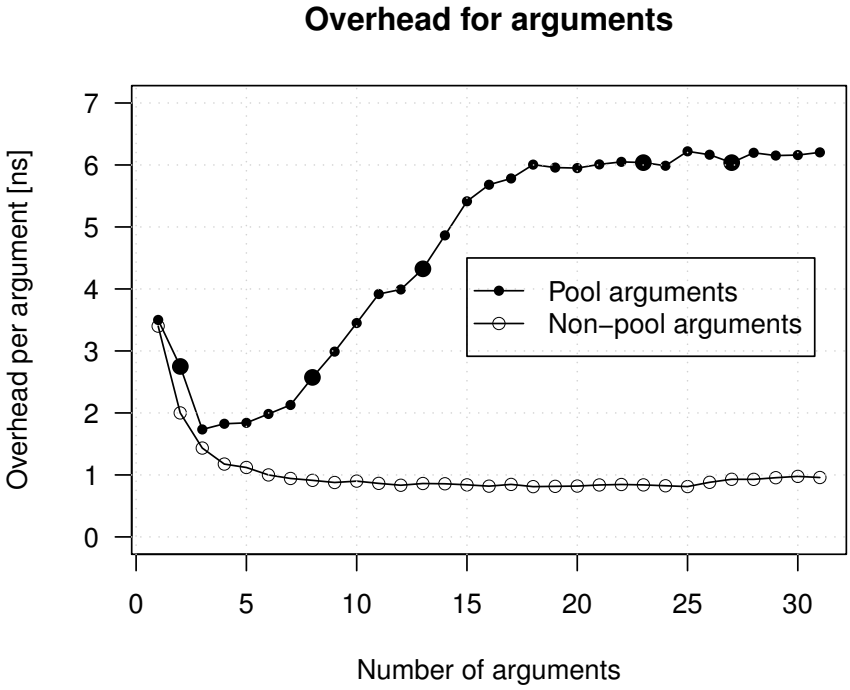



Figure 7.7: Overhead of the function dispatch mechanism for a varying number of both pool and non-pool arguments.

arguments, the average cost per argument decreases quickly to slightly less than $1ns$ when the number of arguments is increased. Pool arguments have a higher overhead due to the conditions that need to be checked (see Figure 7.3).

### Trace Vector Generation

The trace vectors that store the loop bounds of the previous execution of a function are generated when the non-optimized version of a function is executed (See Section 7.1.3). This incurs some overhead as iteration counts are tracked and written to the trace vector. We estimate this overhead by running the pointer-based benchmarks of SPARK00 for only one iteration. During this first iteration, the trace vectors are generated. The execution time of this single iteration is compared to the execution time of a single iteration when using only structure splitting. For small data sets, the time measured is so small that these results are unreliable. Therefore, only data sets for which execution times are larger than $1ms$ are shown in the figures. The

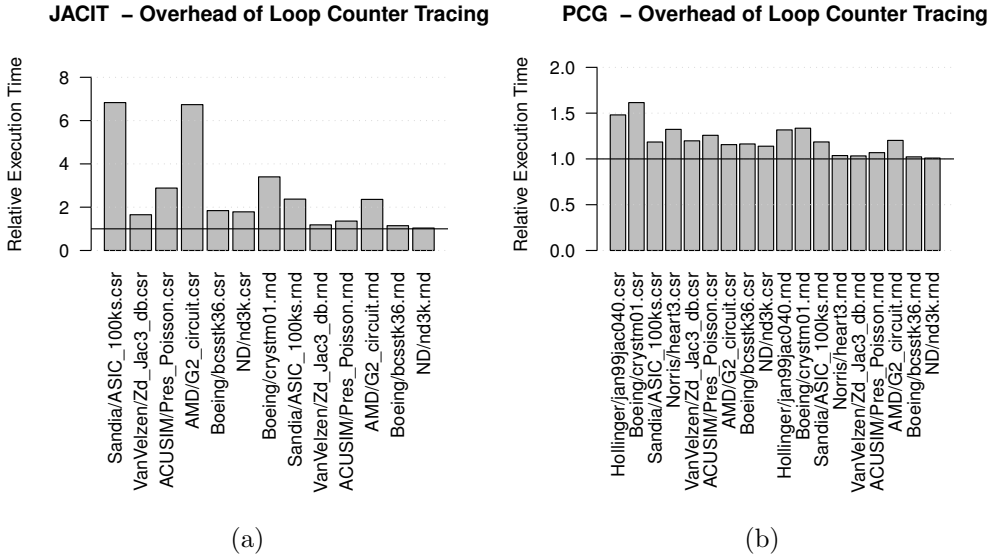



Figure 7.8: Overhead of the loop bound tracing mechanism for the SPARK00 benchmarks JACIT (a) and PCG (b). Only results are included if execution time of a single iteration of the original (split) version was greater than $1ms$. The results for JACIT excludes some matrices, as it cannot use matrices that contain zero elements on the main diagonal.

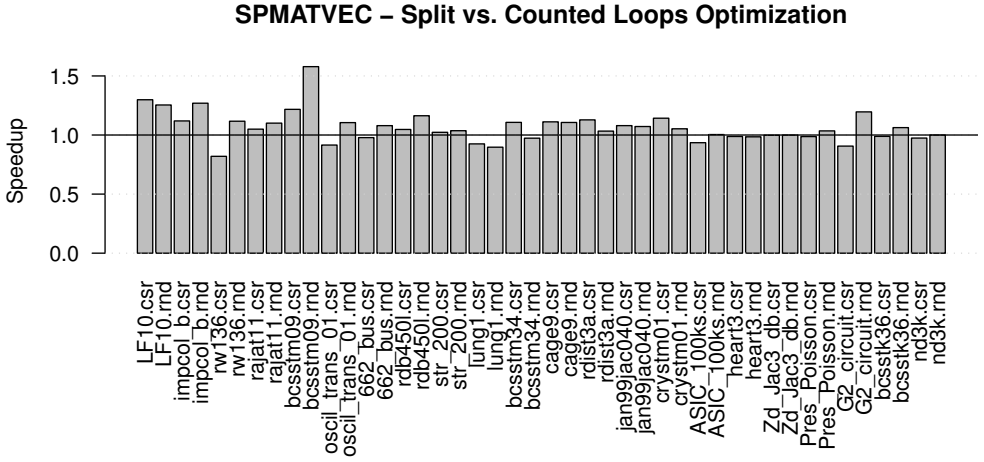



Figure 7.9: Speedups for sparse matrix times vector after applying the complete transformation chain. Results are shown for three different memory layouts per matrix. The input sets are ordered by size from left to right. On large data sets, our transformations do not have a bad impact on performance, while allowing for further optimization.

benchmarks have been executed for two different data layouts, *csr* and *rnd*.

Figure 7.8 shows the relative execution time of JACIT and PCG comparing the version that generates the trace vectors versus the version that only uses structure splitting. The other benchmarks show similar results. Some data sets do not appear in the plot for JACIT, as those matrices contain one or more zero elements on the diagonal and are thus invalid inputs for JACIT. The results are labeled with the memory layout used and are ordered by data layout and size. We observe that the larger data sets suffer relatively less from the tracing mechanism, and that code showing a predictable memory reference pattern has a higher relative overhead. Especially smaller data sets (those that fit into the caches) show a large relative overhead. Because of this fact, for small data sets, focus should be on code optimization, not on data layout, which has already been described in Chapter 3.

The overall conclusion is that the generation of loop bound trace vectors for larger data sets is usually quite small, but for small data sets, the cost can be relatively high.

### 7.2.2 Loop Optimization of Data-Intensive Code

We have seen that structure splitting can yield substantial speedups on these pointer-based benchmarks, if the memory reference pattern is reasonably predictable. This can be achieved by restructuring the memory pools (see Chapter 6). Therefore, we

compare the result of our loop optimizations with the split version. Note that pool restructuring has not yet been integrated in the optimization chain (combined with the control flow optimizations) and therefore restructuring is also disabled for the split version.

Figure 7.9 shows the result of our optimizations on the SPMATVEC benchmark (the other benchmarks show similar results). For this benchmark, the transformations from Section 7.1, except those from Section 7.1.6, have been applied. Note that the data sets are ordered from small to large. In general, especially for the larger data sets, our transformations barely affect performance. Even for smaller data sets, in many cases the overhead of the dispatch mechanism does not result in significantly lower performance.

In some cases, especially for the smaller data sets, some speedups are observed. No data restructuring or code restructuring techniques, as described in Section 7.1.6, have been applied. Therefore, this speedup cannot be caused by a reduction in required memory bandwidth. Inspection of the generated *x86*-64 assembly code shows that the transformed inner loop has a lower instruction count, due to the more efficient translation of the array-based code resulting from the pointer to array-based code transformation described in Section 7.1.5. On small data sets, instruction throughput is a limiting factor, while for large data sets, the instruction count has much less impact.

While there are a few exceptions, on average the performance is comparable to that of the split version, and the generated loop structures are suitable targets for further optimization. This is exactly the goal of these transformations. In itself, it should not have a negative impact on performance, and it should enable traditional array-based optimizations.

## 7.3 Summary

In 1968, Michie proposed memoization [86], a technique that is often used in functional languages. Basically, it is the process in which functions remember the outcome of previous calls to speedup computations. This bears some resemblance to our techniques, in which we detect if control flow did not change with respect to the previous call. Although the work described above shows some relation to our work, the problems targeted are different from ours as we specifically try to bridge the gap between pointer-based and array-based code.

In this chapter, a transformation chain was described which transforms pointer-based codes into array-based codes. Combined with run-time support for the identification of static loop control, we have shown how to obtain a representation that can potentially be optimized by existing techniques. This is a major step forward in the field of optimization for code containing recursive data structures.

The overhead of the run-time checking of preconditions has been determined and, while compared to a regular function there is a measurable overhead, its impact is quite small if any significant computation is done by the function called. The generation of trace vectors can be relatively time consuming for smaller data sets. For

larger data sets, and especially those that cause irregular memory access patterns, we have shown that the tracing overhead is quite small. When applied to the SPARK00 pointer-based sparse matrix computations, it was shown that, even for small data sets, our transformations do on average not degrade performance. Especially, when large data sets and a data layout that causes irregular memory accesses are used, the behavior is very similar in terms of performance compared to the code using structure splitting.

The major contribution of this work is the fact that we transform pointer-based code with data dependent loop conditions to a representation that uses only arrays and simple, counted loop structures. Future work will address the application of existing array-based optimizations. Controlling memory access patterns such as described in Section 7.1.6 will be integrated in our framework enabling more advanced data layout transformations such as described in Chapter 4 and 8. Other directions include parallelization and execution of pointer-based applications on hybrid architectures and GPUs.

# CHAPTER 8

## Data Instance Specific Co-Optimization of Code and Data Structures

For practical reasons, data structure selection and its actual mapping onto hardware do not necessarily match with the logical structure of a problem. Often, a single solution is implemented that is supposed to fit a wide range of problem instances. This, however, is very unlikely to yield optimal performance on the entire range of input sets. Each problem instance has its own characteristics, which are not taken into account by the "one solution fits all" implementation.

An obvious example is sparse matrix representation. Many different storage mechanisms have been proposed, each of which has its applications for specific instances of a more general problem, for instance, solving a sparse system of linear equations. This diversity in implementations, each of which is tailored to specific properties of a problem at hand, is a nightmare from a code management point of view. For each different class of problems, new code should be written. In practice, sub-optimal performance will be the result of this trade-off between implementation effort and efficiency.

Many proposals to optimize irregular and sparse applications have been made recently. Most, if not all, of these proposal rely heavily on extensive run-time analysis [68, 104, 106]. Although these techniques themselves might work very well, they suffer from the fact that the overhead incurred by these run-time mechanisms must be amortized over multiple runs or iterations. To optimize specifically for each input set, a code using these run-time mechanisms is generally not feasible. So in order to allow data instance specific optimizations, other mechanisms are needed.

## 8.1 Aggressive Two-Phase Compilation

In this chapter, we propose a very ambitious and aggressive compilation trajectory to overcome the issues described above and to allow data instance specific optimization. Basically, our approach relies on the fact that for large scale simulation codes, like circuit simulation, structural mechanics, computational fluid dynamics, etc., the reference codes which are used have a long life cycle [60, 82, 99, 100, 102]. Although these codes employ libraries to exploit specific architectural features, in general, it is a major and error-prone task to rewrite these codes and optimize them for a specific problem at hand.

In our approach, these codes will be aggressively analyzed, both at compile and run-time, and they will be automatically expanded into a form which allows compiler optimizations to be much more effective. Also, other compiler optimizations are enabled, which optimize these codes for specific problem instances. We envision that this whole transformation chain is split into two parts. The first part consists of compile and run-time analysis, combined with iteration space expansion and will happen at the vendor/code owner's site. So even before the code is shipped to the customer, the code is prepared, instrumented and expanded. Because this is done on a per customer basis, the amount of time it takes to prepare this code can be considerable. In this phase, representative data sets can be used to identify access patterns that are likely to be useful for restructuring later. In the second phase, after the code is installed at the customer's site, "back-end" compiler optimizations take place, which do not only optimize for specific architectural features of the computing platform, but also take the specific characteristics of the problem to be solved into account. In the second phase, the overhead incurred by these compiler optimizations should be minimized.

One might wonder why the code resulting from the first phase cannot be directly provided by the code owner. However, if this approach would be taken, the code owner must maintain multiple versions of the code, as different problem domains show different data usage patterns. A major advantage of our approach is that the code owner only has to maintain one reference code and not multiple versions of the codes for the different customers. Note that the different versions of the code, which otherwise have to be maintained, differ in an essential way, in the sense that they are based on different data structure choices. Therefore, this effort would go far beyond common practice, in which the code owners just have to maintain differently configured versions of its code base for the different customers.

In this chapter, we mainly describe the first phase of this compilation chain, where we specifically describe the way the code could potentially be expanded. This expansion is based on the notion of *sublimation*, in which data structures used are being embedded into *enveloping data structures*, such that proper data dependence analysis can be performed in the second phase. In order to enable this sublimation as a preparatory phase, the pointer-based codes are being transformed into (indirect addressed) array-based codes using pool allocation and structure splitting. Although this in itself is a challenging problem, recent results show that this can be solved using compile-time techniques [35, 48, 73, 75, 76, 115, 116]. This phase will be extended

by automatically generated compiler instrumentation (for tracing memory pool accesses [115]), which uses run-time information to identify regions in which indirections are referring to distinct objects. This information is used in the expansion phase to further eliminate indirect addressing, yielding indirection free, array-based code which can therefore can be analyzed by standard compiler optimization techniques. This representation of the code is then analyzed at the customer's site together with the characteristics of the specific problem instance to be solved to obtain a problem/architectural optimized implementation of the code. The last phase is based on compilation techniques that optimize dense codes together with a non-zero pattern specification into an optimized sparse implementation [25, 26].

Within the scope of this chapter, we will mainly concentrate on the sublimation/expansion process. As already described, the overall compilation chain is very ambitious, but it is our belief that such an approach is needed in the future to tackle the problems faced when implementing data specific optimizations.

In Chapter 6, we have seen how pointer-linked data structures can be rewritten to a split representation that can be reordered at run-time. Subsequently, we described in Chapter 7 how this can be transformed into an array-based representation and how it can be determined if loop conditions may change. In this Chapter, we use this array-based representation, and explain *sublimation*, (the technique to embed data structures into an enveloping data structure) in Section 8.2. We have applied sublimation to three sparse matrix kernels, which are *sparse matrix vector multiplication, Jacobi iteration* and a *direct solver* using an LU-factorized matrix. The derivation using sublimation is done in Section 8.3. The kernels have been optimized using a variety of different input matrices and the results are presented in Section 8.4, which also evaluates and describes the overhead of the compilation chain in the second phase. Section 8.5 summarizes this chapter.

## 8.2 Sublimation

The basic idea behind sublimation is to transform a code using indirect addressing into a dense code, thereby eliminating the occurrence of indirect addressing entirely. So for example, we want to transform a code using compressed row format-based to represent sparse matrices into a dense code using compiler transformations. In this section, we describe how this could be achieved by first restructuring access to arrays to conform to a common access pattern. By transferring the indirection from the loop body to the header, followed by an expansion of the iteration space, a regular intermediate code is obtained.

Prototypes of these techniques have been implemented [113, 125]. They address either simple indirect addressing-based loops or nicely nested pointer traversal loops. These implementations have not yet been extended to use pool-allocated data structures. For the sake of a concise description, we describe these transformations using generic code samples, as the codes generated by the prototype implementations contain additional initialization code which unnecessarily complicates the explanation. An example of real output generated by our transformation process can be found in

Section 8.6, which shows the output of an automatically transformed sparse matrix vector multiplication kernel.

Note that sublimation is part of the first phase of the compilation process. Therefore, the overhead needed when implementing the transformations as described in this section is only incurred once. In the second phase, the data instance specific optimizations will have to be much more efficient.

The example codes all use an array-based representation. In Chapter 7, we have seen that type-homogeneous data structures that are pool-allocated can be rewritten using an indirection array-based style. It also describes a method to detect invariant loop traversal patterns which can be replaced by counted loop structures. This invariance is used to obtain a countable iteration space for data dependent loop structures whose conditions are proved to be constant with respect to a specific program region. In the remainder of this chapter, we will express our transformations using the indirection array-based version of the pointer-based codes, and thus assume that pointers and data dependent *while* loops have been eliminated using the techniques from Chapter 7.

The analysis and code generation presented in this section which eliminate indirection consists of three (sub)phases. First, data access functions are determined and a common injective access function is selected, possibly by extending the dimensionality of an access function in order to obtain injectivity. In the second phase, the indirection is transferred to the loop header. Subsequently, in the third phase, the irregular iteration space is embedded into a containing iteration space, with known loop bounds.

### 8.2.1 Data Access Restructuring

Consider the following loop structure:

```
for (i=0; i<n; i++ ) {
   ...A[i]...
   ...B[C[i]]...
}
```

In this loop, two arrays are accessed using the iteration counter $i$ and the index array $C$, which basically is a function of the iteration counters. An obvious choice to restructure data in this loop is to restructure array $B$ in order to follow the same access pattern as $A$.

```
for (i=0; i<n; i++ ) {
   ...A[i]...
   ...B'[i]...
}
```

Where $B'$ is defined as: $\forall i, 0 \leq i \leq n : B'[i] = B[C[i]]$. While this *gather* operation is the most obvious choice for restructuring, we will focus on another choice, which is

sublimation of array $A$, by changing its regular access pattern into the same irregular access pattern of $B$. In this case, the code is transformed into:

```
for (i=0; i<n; i++ ) {
   ...A'[C[i]]...
   ...B[C[i]]...
}
```

In the resulting code, both the arrays are accessed using the same access pattern. Of course, the restructuring of the arrays must meet certain criteria in order to be valid. All values used in the original array must be preserved in the target array and in case an array is written to, data should not be duplicated.

Generally speaking, let $f(I)$ and $g(I)$ denote the new and original access functions, respectively. An access function is defined as a function that maps a point from the iteration space into an integer offset. In the example above, $f(i) = C[i]$ (access using index array) and $g(i) = i$ (the iteration counter). The following condition must hold if $f(I)$ is used to access data that is being read: $\forall I, J : I \neq J \rightarrow f(I) \neq f(J)$ (thus, $f$ is an injective function). If $f(I)$ is used to index an array that is written to, then the injectivity of $f(I)$ is not sufficient. In that case, the original access function must also be injective, in order to avoid duplication of data originating from the same location. For writes, the following condition must hold: $\forall I, J : I \neq J \rightarrow (f(I) \neq f(J) \wedge g(I) \neq g(J))$ (both $f$ and $g$ are injective functions).

### 8.2.2 Identifying Injective Functions in Code

The notion of sublimation that was explained in the previous section was based on a simple regular access pattern combined with an indirect access pattern. In general, the simple regular access pattern can be extended to any injective access pattern derived from loop counters. Such patterns naturally arise in counted loop structures. From such loop structures, the iteration space and a total ordering of its traversal can be determined at compile-time. On this total order $T$, a *bijective* mapping $h : T \rightarrow \mathcal{Z}^n$ to the iteration space can be defined. Using this bijective mapping, all of the techniques described above are generalized to multi-dimensional iteration spaces.

Within a loop, multiple access patterns can be identified, each of which is a potential candidate to be used as the access function to which other access functions adapt. This adaptation is what the notion of sublimation refers to. While not all access functions are injective, all access functions can be made injective with respect to the containing loop structure by expanding their dimensionality. We will clarify this using an example:

```
#pragma INJECTIVE(k)
for (i=0; i<n; i++ ) {
#pragma INJECTIVE(colIdx[k])
   for (k=start[i]; k<start[i+1]; k++ ) {
     result[i] += M[k] * right[ colIdx[k] ];
   }
}
```

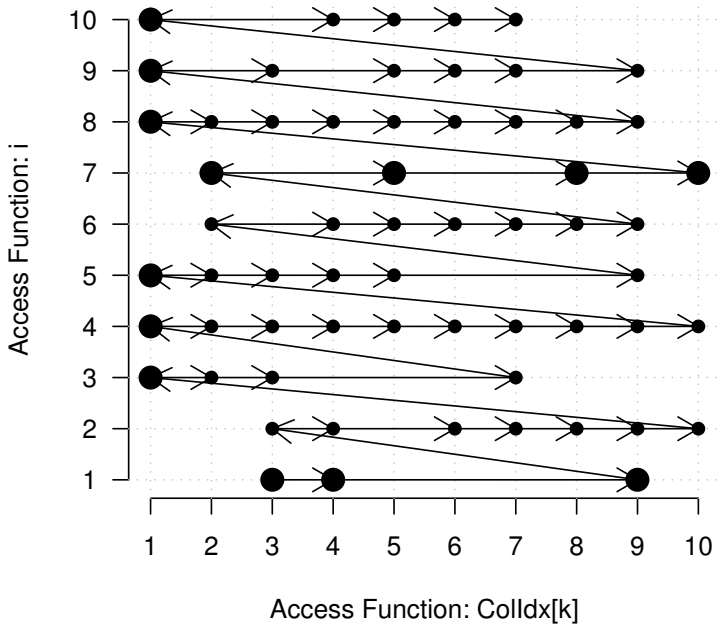



Figure 8.1: $colIdx[k]$ is a non-injective access function. Extending the index expression with $i$ results in an injective access function. The arrows show the order in which the points in the function are visited.

This code computes a sparse matrix vector product. The addressing expressions that can be derived from this code sample are: $i$, $k$ and $ColIdx[k]$. Of these, $colIdx[k]$ is injective with respect to every single iteration of the inner loop and $k$ is injective across the entire iteration space. This can be specified in a directive (which is done in our example) or we can choose to speculate on this property and check injectiveness at run-time. In this example, $colIdx[k]$ is not injective across the entire iteration space. By extending the access function using one of the dimensions of the iteration space, a new, injective function is obtained. In our example, the new injective function is $f(i,k) = (i, colIdx[k])$, see Figure 8.1. As a result, the following loop structure is obtained (the *#pragma* statements are left out):

```
for (i=0; i<n; i++ ) {
   for (k=start[i]; k<start[i+1]; k++ ) {
     result[i] += M'[i, colIdx[k]] * right[ colIdx[k] ];
   }
}
```

Note that this resulting code is *never* directly executed. It only serves as an intermediate code. In this code, `M'[i, colIdx[k]] = M[k]`.

### 8.2.3 Eliminating Indirect Addressing in the Loop Body

Irregularity can be present both in the loop header as well as in the loop body. With irregularity, we mean any property that cannot be statically determined. This includes, data dependent loop conditions and unpredictable memory reference patterns. In our example, the lower and upper bound of the inner loop are data dependent and thus irregular. The inner loop is a counted loop, which in itself defines an injective function. This injective property can be used to transfer the irregular access that still exists within the inner loops to the loop header. Let the original iteration space be $\Delta$ and let $h$ be an irregular access function within the loop. Then the irregular access function can be transferred to the loop header as follows:

```
for ( I ∈ Δ ) {
   ...A'[h(I)]...
}
```

is transformed to:

```
for ( I' ∈ h(Δ) ) {
   ...A'[I']...
}
```

Applied to our example, the irregular access function defined by the index array expression $colIdx[k]$ is dependent on the inner loop counter, and the access function can be transferred to the loop bounds of the inner loop as follows:

```
for (i=0; i<n; i++ ) {
   for ( q ∈ { colIndex[start[i]], colIndex[start[i]+1],
           ..., colIndex[start[i+1]-1] } ) {
     result[i] += M'[i, q] * right[q] ];
   }
}
```

The irregularity has now been transferred from the loop body to the loop header.

### 8.2.4 Expanding the Iteration Space

The resulting loop header is still data dependent and thus irregular. This form of irregularity can be eliminated by expanding the iteration space to a space that encompasses the entire iteration space using a fixed interval that is large enough to contain all the elements of the original iteration space. The fact that this can be done, relies on the property that statements of the following form do not have any effect:

(1) $A = A + 0$
(2) $A = A * C$, if A is zero.

Therefore, any extraneously executed statements will not change the semantics of programs. Such statements naturally occur in numerically intensive applications, and therefore this method is suitable for large scale simulation codes as mentioned in the introduction.

In general, let $\Delta$ be the iteration space after transferring the indirect access to the loop header, as described in the previous section. Let $\Omega$ be the new iteration space that extends $\Delta$ ($\Omega \supseteq \Delta$). The injective function $g$ is the extended function of the original access function $f$. If $A$ is the result of sublimation on an array, the array $A'$ used in the expanded iteration space is defined as follows: $\forall I \in \Omega : A'[g(I)] = A[f(I)]$ if $I \in \Delta, 0$ if $I \in \Omega \backslash \Delta$.

Applied to our example code, the iteration space can be extended by transforming the inner loop to a counted loop with a range that covers all possible values. The new access function is still $(i, q)$, but $q$ covers a larger range of values.

```
for (i=0; i<n; i++ ) {
   for ( q=0; q<MAX_INT; q++ ) {
     result[i] += M''[i, q] * right[q] ];
   }
}
```

Note that this code is an intermediate code, and therefore is never executed. The loop bounds here are taken very conservatively. Using directives or results from other analyses, a smaller iteration space could be used. Eventually, this intermediate is recompiled and proper loop bounds are generated. Note that the definition of $M'$ needs to be changed into $M''$, in order to specify the zero elements:

$$
\begin{array}{l}
\forall (i, q) : \\
\quad \text{if } q \in \{colIndex[start[i]], \ldots, colIndex[start[i+1]-1]\} \\
\quad\quad M''[i, q] = M'[i, q] \\
\quad \text{else} \\
\quad\quad M''[i, q] = 0
\end{array}
$$

## 8.3 Application of Sublimation to Pointer-based Matrix Kernels

We have applied sublimation on three sparse matrix kernels, that use orthogonally linked lists to store the matrix data. The three kernels considered are *sparse matrix multiplication, sparse Jacobi iteration* and a *direct sparse solver* using an LU-factorized matrix, and are taken from SPARK00 [114]. These kernels have been transformed to an array-based representation using the transformations described in Chapter 7. On each of these kernels, sublimation is applied, causing the array representation of the matrix values to be extended. Although all programs use the orthogonally linked list representation for sparse matrices, the difference in traversal patterns may lead to completely different implementations for different kernels. In this section, the transformation of each of the three kernels is described and characteristic features of each of these kernels are explained.

```
k=0;
for (i=0; i<n; i++ ) {
   for (j=0; j<limj[i]; j++ ) {
     result[i] += M[k] *
              right[colIdx[k]];
     k++;
   }
}
```

(a) Pointers transformed to arrays

```
for (i=0; i<n; i++) {
   for (k=start[i];
         k<start[i+1]; k++) {
     result[i] += M'[i,colIdx[k]] *
              right[colIdx[k]];
   }
}
```

(c) Sublimation of M to M'

```
for (i=0; i<n; i++) {
   for (k=start[i];
         k<start[i+1]; k++) {
     result[i] += M[k] *
              right[colIdx[k]];
   }
}
```

(b) Injective inner loop

```
for (i=0; i<n; i++) {
   for (q=0; q<INT_MAX; q++) {
     result[i] += M''[i,q] *
              right[q];
   }
}
```

(d) Transferred and expanded loop bounds

Figure 8.2: Starting point for code analysis and sublimation of sparse matrix vector multiplication.

### 8.3.1 Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication is an important kernel in many applications. Figure 8.2 shows the code samples while the code is being transformed. We start with the array-based code resulting from the pointer to array conversion. In this code, the variable $k$ is increased in the inner loop body. Therefore, $k$ defines an injective access pattern. The values that $k$ takes for each inner loop can be determined during the pointer to array conversion and results in a loop structure where the bounds of $k$ are defined in the loop header.

We pick the access pattern $colIdx[k]$ to be used for sublimation and redefine $M$ accordingly. As explained in Section 8.2.2, access functions that are not injective can be made injective by extending them using a dimension from the iteration space. In this case, it is extended using the iteration counter $i$. Injectivity must be determined either by using directives or by run-time analysis.

Subsequently, the indirection is moved to the loop header, after which the iteration space is expanded, using the property that the access function is injective with respect to the inner loop. The resulting dense intermediate code will be optimized later when the actual data set has been loaded at run-time.

```
k = 0;
for (h=0; h<hlim; h++) {
   for (i=0; i<ilim[h]; i++) {
     x_2[h] -= M[k]*x_1[colIdx[k]];
     k++;
   }

   for (j=0; j<jlim[h]; j++ ) {
     x_2[h] -= M[k]*x_1[colIdx[k]];
     k++;
   }
   x_2[h] = x_2[h] / diag[h];
}
```

(a) Pointers transformed to arrays

```
for (h=0; h<hlim; h++) {
   for (k=start[h];
         k<start[h+1]; k++) {
     x_2[h] -= M[k]*x_1[colIdx[k]];
   }
   x_2[h] = x_2[h] / diag[h];
 }
```

(b) Fused injective inner loop

```
for (h=0; h<hlim; h++) {
   for (k=start[h];
         k<start[h+1]; k++) {
     x_2[h] -= M'[h,colIdx[k]]*
               x_1[colIdx[k]];
   }
   x_2[h] = x_2[h] / diag[h];
}
```

(c) Sublimation of M to M'

```
 for (h=0; h<hlim; h++) {
    for (q=0; q<INT_MAX; q++ ) {
      x_2[h] -= M''[h,q]*x_1[q];
    }
    x_2[h] = x_2[h] / diag[h];
 }
```

(d) Transferred and expanded loop bounds

Figure 8.3: Starting point for code analysis and sublimation of Jacobi iteration.

### 8.3.2 Jacobi Iteration

Figure 8.3 shows the code for Jacobi iteration while it is being transformed. Jacobi iteration has the interesting property that it consists of two inner loops. These loops originate from the pointer-based code, where elements before and after the diagonal are traversed in separate loops, while storing the diagonal entry to a temporary variable. In the array-based version shown here, the diagonal entries are stored in the array *diag*. Therefore, the loops can now be merged into a single loop. Similar as in the sparse matrix multiplication code, $k$ is injective and can be transferred to the inner loop by storing the lower and upper bounds per execution of the inner loop.

The sublimation process and iteration space expansion follow the same pattern as previously. It should be noted, though, that the definition of $M''$ will not include entries from the main diagonal, as these are separately stored in the array *diag*. While looking similar to sparse matrix multiplication, the missing diagonal might lead to different optimization choices in the optimization back-end.

### 8.3.3 Direct Solver

The results of the transformation steps on the direct solver are shown in Figure 8.4. Characteristic for the kernel is the use of two different index arrays, one using row indices ($rowIdx$) upon column-wise traversal of the matrix and one using column indices ($colIdx$) upon row-wise traversal. The lower triangle of the matrix is traversed column-wise (first loop), the upper triangle is traversed row-wise (second loop). Indirect access is both present in reads of arrays and in writes.

In the code, the data has been segmented in two different arrays, $M$ and $M2$, representing the lower triangle and upper triangle, respectively. $k$ and $k2$ define an injective access pattern, and are put in the inner loops with corresponding lower and upper loop bounds. In the first loop, the variable $i$ is an induction variable and is replaced by $g+1$. In the first loop, $M$ is sublimated using the access function $rowIdx[k]$, while in the second loop $M2$ is sublimated using the access pattern $colIdx[k2]$. After transferring indirect addressing and expanding the iteration space, the intermediate dense code specifies two dense matrices, each representing the lower and upper triangle of the input matrix. Similar as in Jacobi iteration, the main diagonal is not part of the $M''$ or $M2''$, but is stored separately in the array *pivot*.

## 8.4 Experiments

The sparse matrix kernels, on which sublimation has been applied, have been optimized and executed using a variety of matrices from Davis's *University of Florida Sparse Matrix Collection* [38]. Not all data sets that are used in the sparse matrix multiplication are used in Jacobi iteration because input matrices for Jacobi iteration cannot contain zero entries on the diagonal. For the direct solver, the matrices have been LU-factorized prior to running the program. Matrices taking excessive time to factorize have not been used.

The dense intermediate codes derived in the previous section are compiled to a

```
I = 1; k = 0;
for (g=0; g<glim; g++) {
   Temp = Intermediate[I];
   Temp = Temp / pivot[g];
   Intermediate[I] = Temp;
   for (m=0; m<mlim[i]; m++) {
     Intermediate[rowIdx[k]] -=
             Temp * M[k];
     k++;
   }
   I++;
}

I2 = Size; k2 = 0;
for (h=0; h<hlim; h++) {
   Temp = Intermediate[I];
   for (n=0; n<nlim[h]; n++) {
     Temp -=
       M2[k2]*Intermediate[colIdx[k2]];
     k2++;
   }
   Intermediate[I2] = Temp;
   I2--;
}
```

(a) Pointers transformed to arrays

```
for (g=0; g<glim; g++) {
  Temp = Intermediate[g+1];
  Temp = Temp / pivot[g];
  Intermediate[g+1] = Temp;
  for( k=start[g]; k<start[g+1]; k++ ) {
    Intermediate[rowIdx[k]] -=
              Temp * M[k];
  }
}

I2 = Size;
for (h=0; h<hlim; h++) {
  Temp = Intermediate[I2];
  for( k2=start2[h]; k2<start2[h+1]; k2++)
   {
    Temp -=
      M2[k2]*Intermediate[colIdx[k2]];
  }
  Intermediate[I2] = Temp;
  I2--;
}
```

(b) Injective inner loops

```
for (g=0; g<glim; g++) {
   Temp = Intermediate[g+1];
   Temp = Temp / pivot[g];
   Intermediate[g+1] = Temp;
   for( k=start[g]; k<start[g+1]; k++ )
{
     Intermediate[rowIdx[k]] -=
             Temp * M'[g,rowIdx[k]];
   }
}

I2 = Size;
for (h=0; h<hlim; h++) {
   Temp = Intermediate[I2];
   for( k2=start2[h];
       k2<start2[h+1]; k2++) {
     Temp -= M2'[h,colIdx[k2]] *
             Intermediate[colIdx[k2]];
   }
   Intermediate[I2] = Temp;
   I2--;
}
```

(c) Sublimation of arrays M and M2

```
for (g=0; g<glim; g++) {
  Temp = Intermediate[g+1];
  Temp = Temp / pivot[g];
  Intermediate[g+1] = Temp;
  for( q=0; q<INT_MAX; q++ ) {
    Intermediate[q] -= Temp * M'[g,q];
  }
}

I2 = Size;
for (h=0; h<hlim; h++) {
  Temp = Intermediate[I2];
  for( r=0; r<INT_MAX; r++) {
    Temp -= M2''[h,r]*Intermediate[r];
  }
  Intermediate[I2] = Temp;
  I2--;
}
```

(d) Transferred and expanded loop bounds

Figure 8.4: Starting point for code analysis and sublimation of a direct solver.

data set-specific implementation of the kernel. In our experiments, we use the MT1 compiler as our back-end [25]. MT1 compiles a dense specification together with the specification of the non-zero patterns and produces a data set-specific optimized implementation. The access patterns can be obtained by instrumenting the code obtained by the methods presented in Chapter 7. All experiments have been run on Intel Core 2 Duo 2.33GHz system with 2GiB of main memory, running Mac OS X 10.6.2. The programs are compiled using GCC 4.2.1 using the options '-O3 -ftree-vectorize'.

First, we will present the results of the recompiled kernel that uses the dense code as input and the access patterns as defined by the specific input matrix. Next, the overhead of the dynamic code-generation is assessed. Note that in the first phase of our overall compilation process, the overhead can be substantial. However, when transforming the intermediate (dense) code into a data instance specific code, which belongs to the second phase of the overall compilation process, the overhead should be minimized.

### 8.4.1 Results on Sparse Matrix Kernels

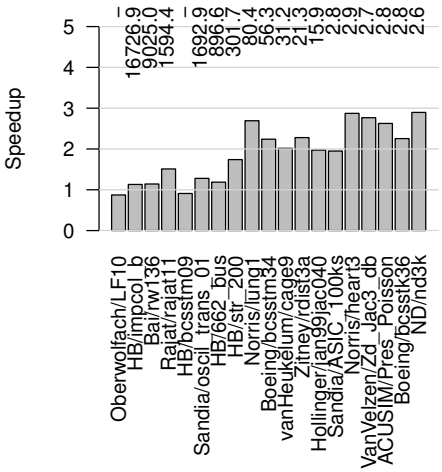
As soon as the data is loaded from the input file and it is determined that the access pattern of the kernel will remain static (as described in Chapter 7), the code resulting from sublimation together with a definition of the access patterns is compiled. The overhead is measured separately, as this is constant for each kernel and data set combination. Including the overhead in the kernel execution time yields arbitrary results, as by increasing the number of iterations, the overhead per iteration can be made as small as desired. Therefore, we will show more details about the overhead in the next section, and here we will focus on the performance of the kernel itself.

Figure 8.5 shows the speedups obtained by running the optimized kernels. In these figures, the data sets are sorted by increasing size from left to right. For each of these kernels, we can observe that the restructuring methods are most suitable for the larger data sets, while the optimizations do not negatively effect performance for the small data sets, in general. The bars are annotated with the break-even points ($\times$1000 iterations). In the cases where the optimized version is slower, a break-even point does not exist and this is denoted with a dash.

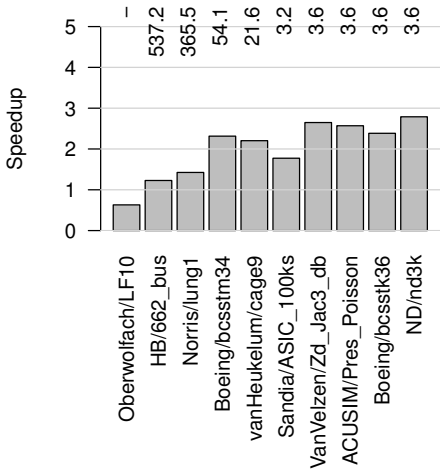
As we can see, the speedups can be substantial. Although the data structure layout for the original kernels have been optimized and take into account the sparsity of the data, they have not been optimized for specific non-zero structure information. As a result of our transformations, these original codes are compared to kernels which are specifically optimized for a particular input matrix, and thus both the code and data layout are optimized for that specific data instance.

### 8.4.2 Overhead

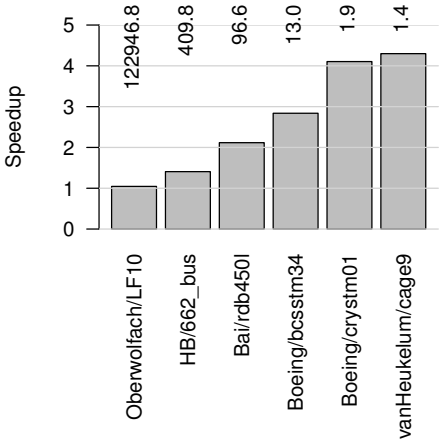
While the application of sublimation is a compile-time analysis and transformation, which generates the dense intermediate code, the data set specific compilation is deferred to run-time. At run-time, the final re-targeting of the code using input data set dependent access patterns is performed before the kernel is executed.




(a) Sparse matrix vector multiply

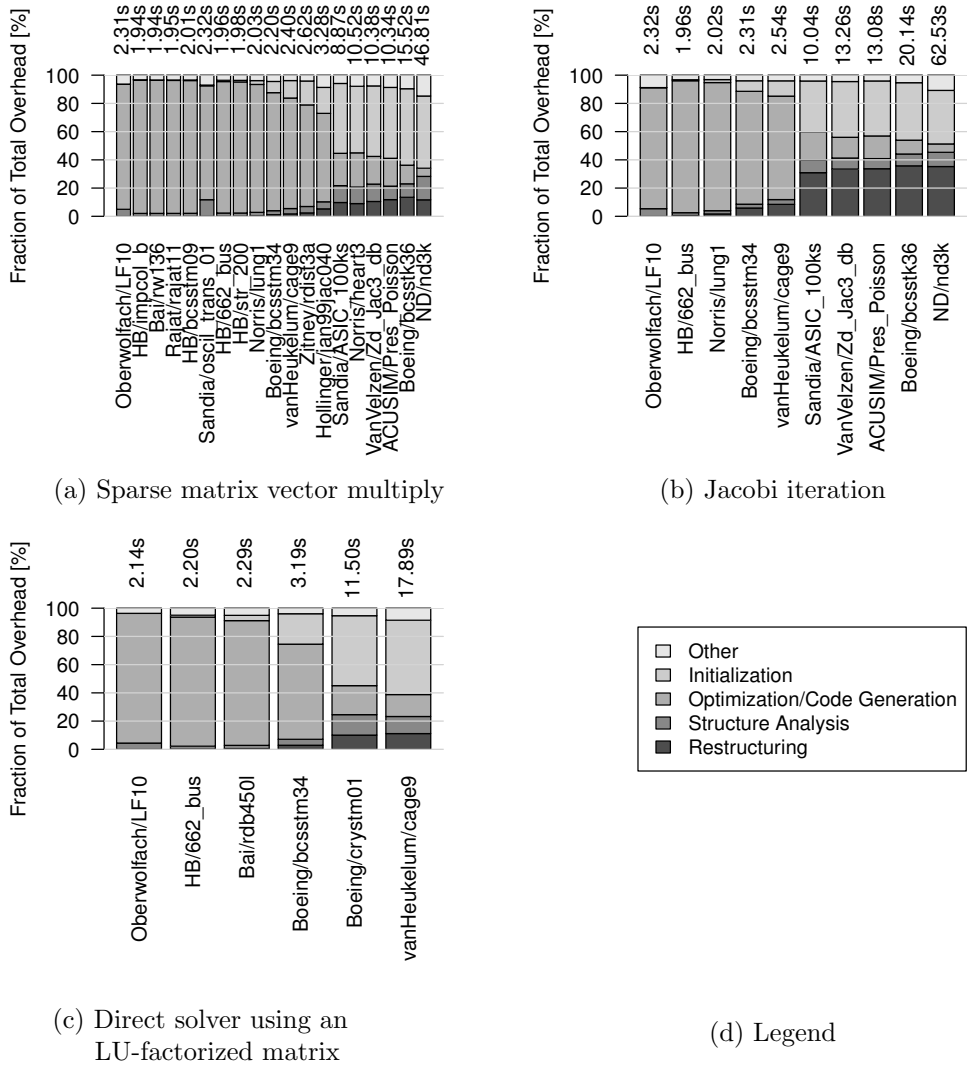



(b) Jacobi iteration




(c) Direct solver using LU factors

Figure 8.5: Speedups obtained on the data set specific optimized kernels. The annotations show the break-even points (×1000 iterations).




(a) Sparse matrix vector multiply

(b) Jacobi iteration

(c) Direct solver using an LU-factorized matrix

(d) Legend

Figure 8.6: Contribution of different phases to the overhead of run-time compilation of the dense intermediate with data set dependent access pattern to a data set specific implementation. The time plotted at the top of each bar is the total time spent in run-time compilation.

Figure 8.6 shows the overhead for the run-time compilation for the kernels with their different input data sets. For the smaller data sets, it is clear that the majority of the time is spent in optimization and code generation (this is the generation of a shared library). This is caused by the fact that this is a fairly constant factor, as this consists of compiling the code emitted by MT1 (using for example GCC) into a binary. More complicated code could be emitted for larger data sets, but this only results in a relatively small increase in compilation time (observed times for this phase range from $0.27s - 4.37s$). For the larger data sets, restructuring, structure analysis and initialization (loading data using the restructured data layout) are the dominating factors.

## 8.5 Summary

We proposed and described a two-phase approach to the optimization of applications. The first phase consists of compile-time analysis in which data used in the application is embedded in enveloping data structures. This is driven by a technique we call *sublimation*, which forces data to be laid out in memory using a common and appropriate access function. The resulting code is an intermediate code which is not directly executed. In the second phase this intermediate is compiled together with actual data and an instance specific optimized code is generated.

We have described how these optimizations can potentially be applied to codes including pointer-based data structures. The subsequent optimizations in the first phase are all based on array-based codes. Using three sparse matrix kernels, we have evaluated the potential of the transformation to an enveloping data structure (a dense matrix in this case) and show that considerable speedups can be achieved. The overhead involved in the restructuring during the second phase has been evaluated. For smaller data sets, the final code generation time is dominating. For large data sets, restructuring and initialization of the new data structures after compilation are dominating.

While the experiments show the potential of our two-phase compilation system using sublimation, these experiments are only a limited application of these techniques. We envision applications in which computational intensive parts are compiled into an intermediate code using sublimation, which is compiled together with specific input data to enable even more aggressive optimizations to be applied.

## 8.6 Example Data Instance Specific Code

```
struct {
  doublereal      val_mtca__[41595];
  integer         ind_mtca__[41595],
                  low_mtca__[3534],
                  hgh_mtca__[3534],
                  lst_mtca__;
}               mtca_____;
#define mtca_____1 mtca_____
static integer  c__3 = 3;
static integer  c__1 = 1;
static integer  c__4 = 4;
static integer  c__3534 = 3534;
static integer  c__41595 = 41595;
int mtc_fortran_init__(void)
{
  integer         i__1;
  olist           o__1;
  cllist          cl__1;
  integer         f_open(olist *), s_rsle(cilist *),
                  do_lio(integer *, integer *,
                              char *, ftnlen),
                  e_rsle(void);
  int             s_stop(char *, ftnlen);
  integer         f_clos(cllist *);
  static integer  i___, j___, k___, m___, n___;
  static real     v___;
  static integer  nnz___, tmp____[41595];
  extern int      dini____(doublereal *, integer *,
                    integer *, integer *, integer *,
                    integer *, integer *, integer *);
  static cilist   io___1 = {0, 1, 0, 0, 0};
  static cilist   io___6 = {0, 1, 0, 0, 0};
  mtca_____1.lst_mtca__ = 1;
  o__1.oerr = 0; o__1.ounit = 1; o__1.ofnmlen = 7;
  o__1.ofnm = "mtca.cs"; o__1.orl = 0; o__1.osta = "OLD";
  o__1.oacc = 0; o__1.ofm = 0; o__1.oblnk = 0;
  f_open(&o__1);
  s_rsle(&io___1);
  do_lio(&c__3, &c__1, (char *) &m___,
          (ftnlen) sizeof(integer));
  do_lio(&c__3, &c__1, (char *) &n___,
          (ftnlen) sizeof(integer));
  do_lio(&c__3, &c__1, (char *) &nnz___,
          (ftnlen) sizeof(integer));
  e_rsle();
  if (m___ != 3534 || n___ != 3534) {
    s_stop("Incorrect size", (ftnlen) 14);
  }
```

```
  i__1 = nnz___;
  for (k___ = 1; k___ <= i__1; ++k___) {
    s_rsle(&io___6);
    do_lio(&c__3, &c__1, (char *) &i___,
          (ftnlen) sizeof(integer));
    do_lio(&c__3, &c__1, (char *) &j___,
          (ftnlen) sizeof(integer));
    do_lio(&c__4, &c__1, (char *) &v___,
          (ftnlen) sizeof(real));
    e_rsle();
    ++mtca_____1.lst_mtca__;
    if (mtca_____1.lst_mtca__ > 41595) {
      s_stop("Too many entries", (ftnlen) 16);
    }
    mtca_____1.val_mtca__[mtca_____1.lst_mtca__ - 1] = v___;
    mtca_____1.ind_mtca__[mtca_____1.lst_mtca__ - 1] = j___;
    tmp____[mtca_____1.lst_mtca__ - 1] = i___;
  }
  cl__1.cerr = 0; cl__1.cunit = 1; cl__1.csta = 0;
  f_clos(&cl__1);
  dini____(mtca_____1.val_mtca__, tmp____,
        mtca_____1.ind_mtca__, mtca_____1.low_mtca__,
        mtca_____1.hgh_mtca__, &c__3534, &c__41595,
        &mtca_____1.lst_mtca__);
  return 0;
}

int mtc_callback_mtca00__(doublereal * result,
                          doublereal * right)
{
  integer         i__1;
  static integer  mtctemplin1, row, mtctemplin1___;
  --right;
  --result;
  for (row = 1; row <= 3534; ++row) {
    result[row] = 0.f;
    i__1 = mtca_____1.hgh_mtca__[row - 1];
    for (mtctemplin1___ = mtca_____1.low_mtca__[row - 1];
        mtctemplin1___ <= i__1; ++mtctemplin1___) {
      mtctemplin1 =
        mtca_____1.ind_mtca__[mtctemplin1___ - 1];
      result[row] +=
        mtca_____1.val_mtca__[mtctemplin1___ - 1] *
            right[mtctemplin1];
    }
  }
  return 0;
}
```

# CHAPTER 9

# Mapping Pointer-linked Data Structures to an FPGA: A Case Study

Heterogeneous systems are systems that are composed of different types of processors, possibly with memories that are not directly accessible to all components. For example, an FPGA combined with a regular desktop system can be viewed as a heterogeneous system, but also the combination of a general purpose CPU and a GPU card within a single system is a heterogeneous system. In order to co-optimize code and data for such architectures, both the code and data must first have a representation that is architecture and memory address space independent. Using such a representation, code and data can be segmented and distributed between the different resources.

Pointer-linked structures prevent compilers from finding these segmentations of the input data such that code can run in parallel. Parallelization of those codes is notoriously hard, mainly due to two reasons: pointers can be aliased and pointers are address space dependent. The aliasing problem has been studied in-depth, both for stack and heap-allocated data. Address space-dependence is a more practical problem. In heterogeneous environments, pointers used in one place are not always valid in other contexts and therefore, pointers cannot be exchanged directly. Similar problems are faced if a pointer-linked data structure, that has been allocated on a host machine, is to be mapped onto an FPGA. Some solutions to map pointer structures into an FPGA have been proposed [40, 111]. These solutions demand the use of a specific *programming interface* (API), which ensures properties such as type-safety and immutability of data layout by their design.

The main contribution of the work presented in this chapter is is the presentation of a novel, *completely compilation-based* strategy that allows the execution of code using pointer-linked structures on architectures lacking a globally shared address space,

such as FPGAs, GPUs and the Cell processor. The techniques rely on transforming type-safe subsets of structured data into the address independent representation, as described in Chapter 6 and 7. The resulting code, which is address space-independent can in principle be mapped to any architecture, but can also be used to synthesize custom hardware. Using a sparse matrix multiplication kernel as a case study, we will illustrate the chain of transformations that leads to a specification of an algorithm that is free of indirect memory access. Figure 9.1 shows the transformation chain. In the following case study, the transformations are discussed in more detail. Our compilation chain is based on the LLVM compiler framework (see Chapter 5).

In Chapter 6, we have outlined the techniques and requirements for the restructuring of pointer-linked data structures. This resulted in an address space independent representation, which can also be represented using arrays and object identifiers. Since the type-safe memory pools can be monitored, it is possible to identify regions in the program which can safely be transformed to use counted loops (see Chapter 7). Using sublimation, which has been described in Chapter 8, the code is mapped into a dense intermediate code that does not contain indirect accesses.

In this chapter, this dense intermediate code is mapped to an FPGA using the Daedalus tool-flow [90], which provides an integrated and automated framework for system-level architectural exploration, system-level synthesis, programming, and prototyping of heterogeneous Multi-Processor System-on-a-Chip (MPSoC) platforms. Its process network-based format is a suitable representation for automatic parallelization. Using a process-network based parallelizing transformation, we will derive and evaluate a parallel implementation of a code that originally used pointer-linked data structures.

Using a case study (sparse matrix-vector multiplication), we explain our transformation chain. Section 9.1 describes the compile-time transformations that map a pointer-based code in an indirection free intermediate code. The subsequent data-dependent iteration space restructuring and mapping onto the FPGA platform is described in Section 9.2, which also shows timing results on different realizations of the parallel model. Related work is discussed in Section 9.3, followed by the conclusions and future directions in Section 9.4.

## 9.1 Compiler Support for Indirection-free Code Generation

Sparse matrix multiplication is a common method that is used in many iterative solvers. In this section, we describe the compile-time analyses and transformations that are needed to obtain a regular (and thus predictable) specification from an originally pointer list-based implementation. The starting point is the matrix vector multiplication kernel shown in Figure 9.2a. It computes its result by traversing the sparse matrix row-wise, and uses the stored column position as on offset in the vector operand. Note that variable declarations are left out of the code samples.

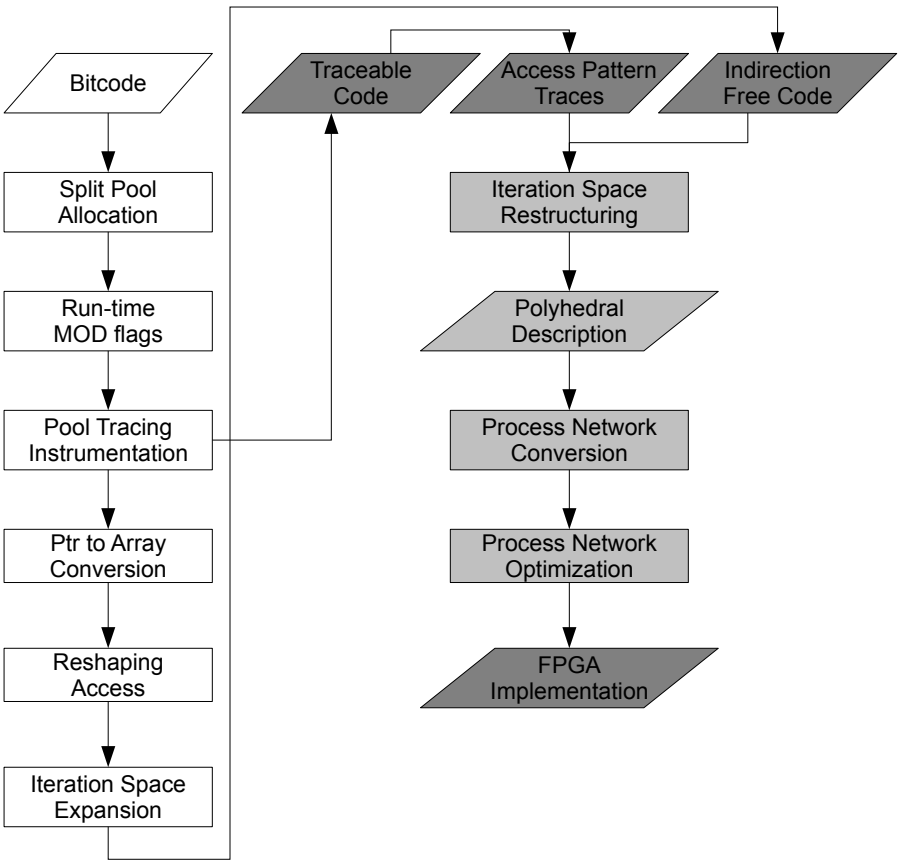



Figure 9.1: Overview of the transformation chain. The white blocks denote pure compile-time phases, the dark gray blocks are run-time components generating information that is used in the run-time data-dependent compilation phases. The run-time data-dependent compilation phases are denoted by light gray blocks.

```
void MatrixMultiplyVec( Matrix *left,
    double *right, double *result )
{
  for(row=1; row<=left->Size;row++){
    result[row] = 0.0;
    pElement = left->FirstInRow[row];
    while(pElement){
      result[row] += pElement->Real *
              right[pElement->Col];
      pElement = pElement->NextInRow;
    }
  }
}
```

(a) Pointer-based code

```
void MatrixMultiplyVec( PoolDescriptor *poolLeft,
    PoolDescriptor *poolElement, Matrix *left,
    double *right, double *result )
{
  unsigned leftAsInt = ptr_to_objid(poolLeft, left);
  double *basePtrFirstInRow =
            poolLeft->basePtr[fieldNo(FirstInRow)];
  unsigned pElement;
  ...

  for(row=1; row <= basePtrSize[leftAsInt]; row++){
    result[row] = 0.0;
    pElement = basePtrFirstInRow[leftAsInt][row];
    while(pElement){
      result[row] += basePtrReal[pElement] *
                right[ basePtrCol[pElement] ];
      pElement = basePtrNextInRow[pElement];
    }
  }
}
```

(b) Array-based code

Figure 9.2: Sparse matrix times vector using linked lists and its corresponding array-based code.

### 9.1.1 Transformation to Pointer Chase-free Code

The transformation chain depends on the structure splitting and restructuring framework that has been presented in Chapter 6 and 7. The split pool allocation, run-time MOD flag tracking, pool tracing and conversion to an array-based representation, depicted in Figure 9.1, are all provided within this framework. Due to the structure splitting transformation, the pointer-based code can be expressed using arrays only, as all data accesses are represented by a $(pool, field, object)$ triplet. Each field has its own data region in the split pool and the start of this data is given by the base pointer for the array-based representation. Figure 9.2b shows the code after conversion to an array-based representation. It shows how the base pointers for the fields are obtained from the pool descriptors and how pointers have become offsets in these arrays. In subsequent code samples, the initialization of base pointers will be left out. All pointers that are loaded from memory pools are already object identifiers, the incoming argument *left* is converted within this function, aided by information from the pool descriptor. In the remainder of the discussion, we assume that each subsequent call will have the same control flow. This can be easily determined from the MOD flag information, which is used at run-time to call the appropriate function. We use multi-versioned code that always includes the original code (i.e. Figure 9.2b). In the case that the MOD flag information indicates that the control flow might be different, the original code will be executed.

Because control flow is guaranteed to be static, that is, data on which the control flow depends has not been modified if the transformed function is called, the loop counters can be traced. Simple, counted loops can be inserted instead (see Chapter 7). In addition, only data that *lives* after execution of the function is relevant to the result of code. Our example needs to access data that is only used for data structure traversal and for indirect access of other arrays. Using the run-time MOD flags, it can also be determined whether access to fields will follow the same pattern and all accesses to pools can be collected in traces. This avoids the latency caused by pointer-chasing loops.

Figure 9.3 shows the code that uses *traces of object identifiers* to access the arrays in the inner loop. Note that only persistent changes to data are relevant for functional correctness. That includes heap data, return values and stack data not local to the function (accessed through a pointer). Local stack data is only valid during the lifetime of the stack frame of the function. Therefore, we determine which changes to data should be persistent after the function returns and mark the associated writes as *necessary operations*. Using dead-code elimination, redundant memory reads needed for data structure traversal are removed. What is left, is a kernel with counted loops and arrays accessed by a trace vector. Naturally, the trace vectors are accessed sequentially. This in itself may already improve performance, as the pointer-chasing is replaced by sequential access to a trace vector. The performance impact of such differences have been described in Chapter 3.

```
void MatrixMultiplyVec( ... )
{
  /**** Initialization left out ****/
  cnt = 0;
  for (i = 0; i < bound; bound++) {
    result[i+1] = 0.0;

    for (j = 0; j < innerBounds[i]; j++) {
      result[i+1] +=
          basePtrReal[basePtrReal_Trace[cnt]]
                * right[ right_Trace[cnt] ];
      cnt++;
    }
  }
}
```

(a) Access by object identifier traces

```
void MatrixMultiplyVec( ... )
{
  /**** Initialization left out ****/
  cnt = 0;
  for (i = 0; i < bound; bound++) {
    result[i+1] = 0.0;

    for (j = 0; j < innerBounds[i]; j++) {
      result[i+1] +=
          basePtrReal[cnt]
                * right[ right_Trace[cnt] ];
      cnt++;
    }
  }
}
```

(b) Pool of *basePtrReal* restructured to sequential access pattern

Figure 9.3: Sparse matrix times vector using (a) object identifier traces and (b) with the restructured array *basePtrReal* that follows a sequential access pattern.

### 9.1.2 Reshaping Memory Access

Given the code from Figure 9.3, we can define an iteration space whose dimensions are determined by the iteration counters of the loop structures. In our example, we have a two-dimensional iteration space with iteration counters $i$ and $j$. Within these loops, we have four addressing expressions: *cnt*, $basePtrReal\_Trace[cnt]$, $right\_Trace[cnt]$ and $i+1$. We will only consider the expressions used for indirect access to arrays.

A bijective mapping between $(i, j)$ and *cnt* can be computed. Figure 9.4a gives an example of a mapping that could be generated from this function. In addition, the trace $basePtrReal\_Trace$ can be analyzed at run-time, and it can be determined if this addressing expression is injective, by verifying the following definition:

$$\forall I, J(I \neq J) \rightarrow basePtrReal\_Trace[I] \neq basePtrReal\_Trace[J].$$

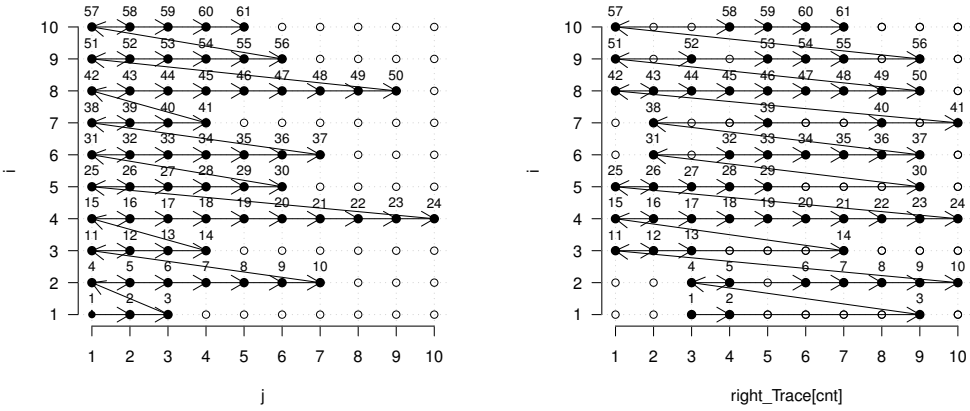
In Chapter 6, we have already seen that memory pools can be permuted, and if this is done using the permutation defined by the trace on $basePtrReal$, the reordered data can be accessed simply by using *cnt* instead of $basePtrReal\_Trace[cnt]$.

Figure 9.4b shows the relation between *cnt* and $(i, right\_Trace[cnt])$. Clearly, $right\_Trace[cnt]$ is not an injective mapping, as multiple values for *cnt* are mapped to the same point when projecting on the axis corresponding to $right\_Trace[cnt]$. Extending the mapping function with the outer loop counter $i$ provides an injective mapping (each point in the figure carries a unique label). Any injective mapping can be used to restructure data. The transformation of regularly accessed data to irregular accessed data (sublimation) has been described in Chapter 8. Application of this technique defines a restructured version of the array $basePtrReal$:

$$basePtrReal\_Restr[F(i, right\_Trace[cnt])] = \begin{cases} basePtrReal[cnt], & \forall cnt(0 \leq cnt < trace\_len) \\ 0, & \text{otherwise} \end{cases}$$

Here, $F$ is a mapping from the two-dimensional logical iteration space to an array index. In these code samples, the mapping used is similar to array access as used in Fortran on multi-dimensional arrays with known sizes: $F(i, j) = i * maxJ + j$, where $maxJ$ is the maximum value found in *right_Trace*.

Figure 9.5 shows the code after restructuring $basePtrReal$. The access pattern to both arrays is now identical, but they are still indirectly accessed. This indirect access can be removed by expanding the iteration space of the inner loop to cover all possible indices. Note that this expansion still happens on the intermediate representation and is *not* necessarily executed. As shown above, the run-time analysis shows that the mapping from *cnt* to $(i, right\_Trace[cnt])$ is injective, and therefore each element is accessed at most once in each iteration of $i$. As a consequence, the iteration space can be extended to include all elements within the range that $right\_Trace$ defines, if this does not affect the result of the computation. In general, this will be true, if the iteration space extension maintains the lexicographical ordering of iterations (which is the case in our example, as $right\_Trace$ contains a monotonically increasing sequence) and the new points in the iteration space do not alter any data (ensured




(a) *cnt* vs. $(i, j)$ (b) *cnt* vs. $(i, right\_Trace[cnt])$

Figure 9.4: Relation of addressing expressions with respect to the iteration space. The arrows denote the traversal order within the iteration space. The annotations on the iterations points are the corresponding values for *cnt*. The open bullets are iterations points that are not visited.

by assuming 0 values in $basePtrReal\_Restr$ for such points). The resulting code that is free of any form of indirect access is shown in Figure 9.5b.

Note that this code serves as an intermediate that is usually not executed, as the iteration space could be really large. Whenever the actual data becomes available, an optimized code can be generated that takes the actual data layout into account. This is described in the next section.

## 9.2 Code Generation and Mapping to an FPGA

After the initial compilation phases (the white blocks in Figure 9.1, we have obtained a traceable and indirection free code, which is used for further optimizations after access pattern traces have been obtained. This section describes the compilation phases used to map the intermediate and indirection free code together with the access pattern traces into an FPGA. First, the iteration space is restructured using the access pattern information, subsequently, we explain how the result is mapped into an FPGA and present our results on the automatic parallelization of the code.

```
void MatrixMultiplyVec( ... )
{
  for (i = 0; i < bound; bound++) {
    result[i+1] = 0.0;
    for (j = 0; j < innerBounds[i]; j++) {
      result[i+1] +=
        basePtrReal_Restr[F(i, right_Trace[cnt])]
             * right[ right_Trace[cnt] ];
      cnt++;
    }
  }
}
```

(a) Restructured data

```
void MatrixMultiplyVec( ... )
{
  for (i = 0; i < bound; bound++) {
    result[i+1] = 0.0;
    for (k = 0; k < sizeOfRight; k++) {
      result[i+1] +=
        basePtrReal_Restr[i*sizeOfRight+k]
             * right[k];

    }
  }
}
```

(b) Restructured data and expanded iteration space

Figure 9.5: Sparse matrix times vector code with restructured data, conforming to the access pattern defined by $right\_Trace[cnt]$ and expanded iteration space, respectively

```
void MatrixMultiplyVec( ... )
{
  for( i = 0; i < outerLoopBound; i++ ) {
    result[i+1] = 0.0;
    for(j = max( halfMB - i, 0 ); j < maxBand; j++) {
      result[i+1] += basePtrReal_Poly[i*maxBand+j] * right[i+1-halfMB+j];
    }
  }
}
```

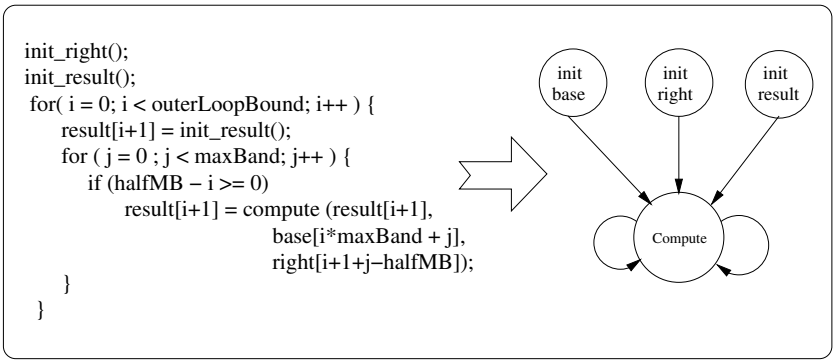
Figure 9.6: The code resulting from iteration space restructuring. The variables $maxBand$ and $halfMB$ are constant values produced by the iteration space restructuring step.

### 9.2.1 Iteration Space Restructuring

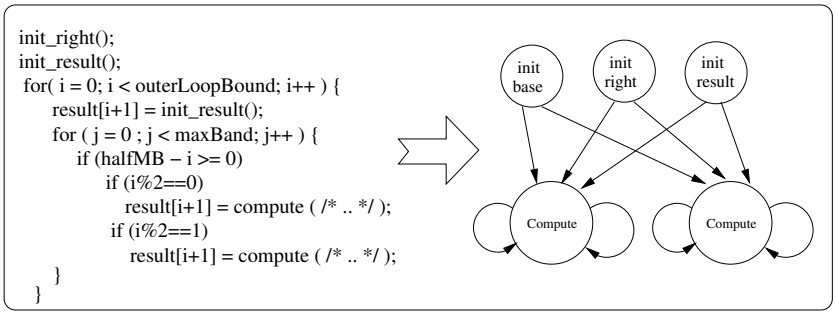
The code in Figure 9.5b is free from indirect address and has regular loops bounds. However, the loop bounds might be very large and many elements might be zero. This all depends on the input data used when calling the function. The trace vector used in Figure 9.5a defines which values are non-zero in the expanded iteration space. Bik and Wijshoff [27] describe algorithms to segment a sparse matrix into regions which can be addressed directly as an affine function of the two loop counters. In our case study, a representation was generated that stores the data in compressed row storage form, but with rows of equal length. This way, the advantage of this data representation is that the offset of each row can directly be determined by a simple affine expression of the loop iteration counters. Figure 9.6 shows the resulting code.

### 9.2.2 Mapping the resulting code to an FPGA

In this section, we show how we map the `MatrixMultiplyVec` kernel from Figure 9.6 onto a Multi-Processor System on Chip (MPSoC) platform prototyped on a Xilinx FPGA. The purpose of this case-study is to show that we can automatically translate kernels with irregular data accesses to code with regular data accesses that can be parallelized and subsequently mapped onto a MPSoC. We use the `pn` compiler [118] to translate the sequential program specification into a parallel one. That is, process networks (PNs) are derived from sequential nested-loop programs in a fully analytical and automated way. However, there are restrictions on the input code: the program must be static and affine, which means that upper/lower bounds of for-loops, array index expressions, and if-conditions are affine combinations of loop iterators and program parameters. Thus, it is not possible to derive, for example, a process network for the linked-list code shown in Figure 9.2b. In the previous sections, however, we have explained how this code can be transformed into another representation as shown in Figure 9.6 that does not expose any linked-list traversal and indirections in the index expressions. This transformation allows the `pn` compiler to derive the corresponding process network, which was not possible before.




(a) Initial process network



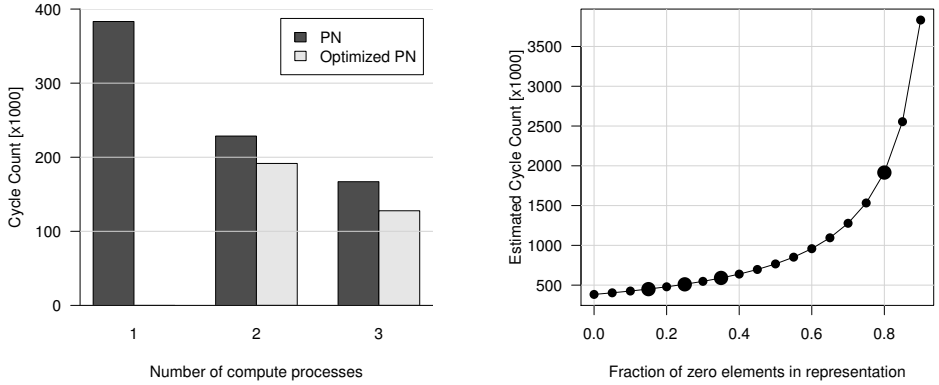

(b) Transformed process network

Figure 9.7: Derivation of different Process Networks. In the experiments, outerLoopBound is equal to 5000, maxBand is equal to 100 and halfMB is equal to 50.

Figure 9.7 shows two different PNs as we have derived them with the `pn` compiler; the input code is shown on the left, and the corresponding derived PNs on the right. The top most figure displays the code for the `MatrixMultiplyVec` kernel as shown in Figure 9.6, but only the data-flow is made explicit for the `pn` compiler. This means that all function input arguments represent reads to particular array elements and the function output arguments are writes to array elements. The actual computation is done inside the function `compute`. It allows the compiler to create a process for each program statement and to automatically derive the control for the FIFO communication for each process. For the input program shown at the top of Figure 9.7, we see that a PN is created that consists of 4 processes; three processes are responsible for initializing data and process `compute` fires a function with the actual computation. It has 2 self-channels, where one is the result of a loop-carried data-dependency on the innermost loop $j$ and the reading/writing from/to array `result`. The other self-channel is used to propagate (or reuse) data from array `right`.

Once the process network is derived, it can be further analyzed and transformed, if necessary. For example, if the performance constraints are not met, then a computationally intensive process can be split up such that multiple instances of that process run in parallel. This is illustrated with an example at the bottom of Figure 9.7, where 2 `compute` processes have been created that execute the same function. We have a compile-time approach to decide how a selected processes can be partitioned best. This is necessary as there are many possibilities to partition a process [85]. The two self-channels of the `compute` process are taken into account, and inter-process communication is avoided if possible. For our example, this means that we create multiple instances of the `compute` process that execute for different iterations of the outermost loop $i$ since the dependencies are carried by the innermost loop iterator $j$. The result is an algorithmic transformation that simply copies the function call such that both compute statements execute the loop iterations in a mutually exclusive way. When the process network is derived, it has 2 `compute` processes without any synchronization and data communication.

Figure 9.8a shows the measured performance results on the ESPAM platform [91]. As mentioned in Figure 9.7, $outerLoopBound$ is equal to 5000, $maxBand$ is equal to 100 and $halfMB$ is equal to 50. The first bar corresponds to the process network of Figure 9.7a that has one compute process. The second group of bars corresponds to the process network of Figure 9.7b that has 2 compute processes. As a baseline, we also did a fully hand-optimized mapping onto the FPGA. The dark bars correspond to the PN as it has been automatically generated by the `pn` compiler, while the light bars correspond to the hand-optimized ones. The transformations introduce additional control (see the modulo statements in Figure 9.7) that can easily be eliminated. The third group of bars corresponds to the PN where the compute process is split up 3 times using the process splitting transformation. In these experiments, all processes are mapped as software one-to-one on MicroBlaze softcore processors, which are point-to-point connected. Thus, the number of processors used is 4, 5, and 6, respectively. After manual optimization of the PNs, the performance increase is linear in the number of processes. This is expected, as the compute processes run in parallel. We can also observe that the additional overhead generated by the au-




(a) Measured performance results

(b) Estimated cycle count for varying degrees of representation efficiency

Figure 9.8: Performance results for sparse matrix-vector multiplication.

tomatic mapping is reasonable. In fact, a further optimization step could potentially eliminate this overhead automatically.

In addition to the measured performance results, we assessed the impact of inefficiencies in the data representation on the performance. This representation is generated during iteration space restructuring, and can contain explicitly stored zero elements. The number of stored zero elements depends on the data layout of the input data set. This leads to the execution of operations that do not affect the result. The results shown in Figure 9.8a are optimal in the sense that their representation is free of zero elements. From this, we can estimate the cycle counts if a suboptimal representation is used. Figure 9.8b shows the cycle counts for the same number of non-zero values, but with a growing fraction of zero elements in the representation, which of course increases when a larger fraction of data consists of zero elements. Thus, finding a good segmentation is a trade-off between the complexity of the iteration space restructuring phase and the performance of the generated code.

## 9.3 Related Work

Diniz and Park use an FPGA as a smart memory engine to reorder memory access of pointer-linked data structures [40]. Their methodology involves the use of a specific programming interface. Thus, the programmer must be aware of the programming model used. Traversal caches, as proposed by Stitt et al. [111], though implemented completely differently, rely on similar foundations we used in our work: recognize pointer-traversal patterns that are static for some time during execution and reorder

accordingly. They recognize their limitation, that their traversal caches must be managed manually, by using their custom software library for maintaining linked data structures.

Our work depends on compile and run-time MOD flags (see Chapter 7) which are used to invalidate traces that define traversal patterns. Therefore, existing software does not need to be rewritten to support our transformations. Where other methods depend on custom libraries to determine traversal patterns and immutability of data on which traversals depend, our framework relies on a combination of compile-time and run-time MOD-flag information.

## 9.4 Summary

Pointer-linked data structures and regular access to data are usually being considered mutually exclusive. In this case study, we have shown that pointer-linked data structures that are used in a type-consistent way, can be transformed into an address space independent representation by applying structure splitting. Access to split memory pools can be traced precisely and data can be restructured by reordering the memory pool. As the split structure representation exposes fields as arrays, these arrays can be optimized by existing methods. In this case study, we used *sublimation* to reshape access in such a way that arrays follow a common access pattern, after which an expansion of the iteration space is applied. This results in an indirection free intermediate code. When actual access patterns are known, a segmentation into convex polyhedra can be made, which we have subsequently mapped to a parallel description using process networks. These networks can be further optimized and we have compared three configurations showing that pointer-based code can be mapped into a parallel hardware implementation when address spaces are not shared.

The choices made in this case study are not the only possibilities. Other applications might offer more restructuring opportunities, but will also show restrictions (e.g. data that cannot be copied efficiently as it mutates constantly).

Not only FPGAs can be targeted using the methods described here. Other architectures, for example GPUs, are also suitable targets as they would certainly benefit if data is restructured. Other applications include automatic distribution of pointer-linked data structures and computations on these structures, which would be a major step forward in research on parallel computing.

# CHAPTER 10

# Conclusions

Data structure selection and its implementation is crucial for the performance of modern applications. This has been true since the first computer programs were written, but the problem has become much more complicated over time. Today, multi-core systems (possibly with heterogenous architectures) are the norm, and this adds another level of complexity. In this thesis, we aimed to bridge the gap between the complexity caused by the use of pointer-linked data structures and existing compiler optimizations. While the gap has not been bridged completely, the following important contributions have been made in this thesis:

- We have presented SPARK00, a set of benchmarks that can be used to assess the impact of irregular code and data layouts, and evaluate the performance of optimization and restructuring strategies.

- Pointers in recursive data structures have been eliminated and are automatically transformed into architecture-independent object identifiers.

- Run-time restructuring of pointer-linked data structures in weakly-typed languages such as C has been described and evaluated.

- Detection of unchanged dependencies of loop conditions has been implemented, which allows for the elimination of long dependence chains in loop conditions.

- A radical, two-phase compilation approach for irregular codes has been proposed. In the first phase, a dense intermediate is derived from the irregular code which in the second phase is compiled, together with the actual data, into a data instance specific intermediate.

These contributions are fundamental to the progress in the field of automatic restructuring and parallelization of code using pointer-linked data structures, an area for which automatic restructuring and optimization has seen limited success over the past years.

Solutions to the problem of programming future computer systems are often sought in the design of new languages and new libraries, as many people seem to believe that automatic compiler parallelization is a dead end. We have shown that this view is too pessimistic, and proved that there are many optimization opportunities in weakly-typed languages such as C. Restructuring of code that uses pointer-linked data structures is feasible, if proper care is taken when writing code. While C and C++ both allow for the violation of type-safety, well-written programs should not do so in order to aid the compiler in optimization.

Looking to the future, we envision that programs can automatically adapt their code and data structures, according to the actual data layout and the available computing resources. Such techniques involve static analysis, run-time monitoring, run-time data instance specific compilation, automatic data migration and identification of parallel sections.

Furthermore, the unified view on data structures should even be taken a step further. Integration of persistent storage into the array-based representation proposed in this thesis is a feasible option. Over the last few years, column-oriented databases have attracted considerable attention and the similarity to split memory pools is striking.

Traditional languages such as C and C++ are here to stay for the foreseeable future. While people may argue that their design and inherent use are unsuitable for automatic parallelization, we have shown in this thesis that, in many cases, automatic parallelization can be successfully applied, Data can be dynamically restructured and we have made the first steps in transparent, automatic restructuring of pointer-linked data structures.

# Bibliography


[1] GCC, the GNU compiler collection. http://gcc.gnu.org.

[2] TOP500 Supercomputing Sites. http://www.top500.org.

[3] High performance Fortran forum: High performance Fortran language specification, version 1.0. Technical Report CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, Texas 77251, 1993.

[4] High performance RDMA protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[5] Co-Array Fortran. http://www.co-array.org, 2010.

[6] Erlang. http://www.erlang.org, 2010.

[7] The LLVM compiler infrastructure. http://llvm.org, 2010.

[8] OpenMP. http://openmp.org, 2010.

[9] Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. *SIGPLAN Not.*, 33(5):269–279, 1998.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[11] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *PLDI '88: Proc. of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 241–249, New York, NY, USA, 1988. ACM Press.

[12] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[13] Randy Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 39(4):75–90, 2004.

[14] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[15] J.W Backus, R.J Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I. Ziller, R.A. Hughes, and R. Nutt. The Fortran automatic coding system. In *Western Joint Computer conference*, pages 188–198.

[16] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[17] C. Bastoul and P. Featrier. Adjusting a program transformation for legality. *Parallel processing letters*, 1(15):3–17, March 2005.

[18] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, September 2004.

[19] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, pages 283–303, Paphos, Cyprus, 2010.

[20] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *PODS '06: Proc. of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 20–29, New York, NY, USA, 2006. ACM Press.

[21] Siegfried Benkner. VFC: The Vienna Fortran compiler. *Sci. Program.*, 7(1):67–81, 1999.

[22] David Bernstein, Michael Rodeh, and Izidor Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.*, 38(9):1308–1313, 1989.

[23] Aart J. C. Bik. *Compiler Support for Sparse Matrices, PhD Thesis*. Leiden University, The Netherlands, 1996.

[24] Aart J. C. Bik. *The Software Vectorization Handbook*. Intel Press, 2004.

[25] Aart J. C. Bik and Harry A. G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31(1):14–24, 1995.

[26] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, 1996.

[27] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM J. Comput.*, 28(5):1576–1587, 1999.

[28] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel Distrib. Technol.*, 2(3):37–47, 1994.

[29] Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, 2002.

[30] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 162–175, 1986.

[31] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: standard template adaptive parallel library. In *SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–10, 2010.

[32] W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, 1999.

[33] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[34] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *PLDI '99: Proc. of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[35] Stephen Curial, Peng Zhao, Jose Nelson Amaral, Yaoqing Gao, Shimin Cui, Raul Silvera, and Roch Archambault. MPADS: memory-pooling-assisted data splitting. In *ISMM '08: Proc. of the 7th international symposium on Memory management*, pages 101–110, 2008.

[36] Stephen Matthew Curial. *Safe Automatic Data Splitting for Linked Data Structures, MSc. Thesis.* University of Alberta, 2007.

[37] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. of the 1969 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM.

[38] T. Davis. The University of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices, submitted to ACM Trans. on Mathematical Software.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[40] Pedro C. Diniz and Joonseok Park. Data search and reorganization using FPGAs: Application to spatial pointer-based data structures. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 207, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Iain S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report RAL 92-086, Chilton, Oxon, England, 1992.

[42] Rudolf Eigenmann and Jay Hoeflinger. Parallelizing and vectorizing compilers, 2000.

[43] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A FORTRAN to C converter. *SIGPLAN Fortran Forum*, 9(2):21–22, 1990.

[44] Agner Fog. The microarchitecture of Intel and AMD CPU's: An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, 2008.

[45] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[46] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff. Experimentally characterizing the behavior of multiprocessor memory systems: A case study. *IEEE Trans. Softw. Eng.*, 16(2):216–223, 1990.

[47] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL '96: Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996. ACM.

[48] Olga Golovanevsky and Ayal Zaks. Struct-reorg: current status and future perspectives. In *Proc. of the GCC Developers' Summit*, pages 47–56, 2007.

[49] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. 1989.

[50] Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in GCC. In *Proc. of the GCC Developers' Summit*, pages 69–92, 2005.

[51] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 150–156, New York, NY, USA, 2002. ACM.

[52] Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420, 1993.

[53] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition*. Morgan Kaufmann, 2006.

[54] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[55] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proc. of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[56] H. Peter Hofstee. Introduction to the Cell Broadband Engine. 2005.

[57] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, 2007.

[58] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI '94: Proc. of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 218–229, 1994.

[59] Yuan-Shin Hwang and Joel H. Saltz. Identifying def/use information of statements that construct and traverse dynamic recursive data structures. In *LCPC '97: Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, pages 131–145, London, UK, 1998. Springer-Verlag.

[60] Flow Science Inc. FLOW-3D. http://www.flow3d.com, 2010.

[61] Intel. Intel architecture optimization manual, 1997.

[62] Intel Corporation. Intel(R) performance tuning utility reference.

[63] W. Jalby, C. Lemuet, and X. Le Pasteur. WBTK: A new set of microbenchmarks to explore memory system performance for scientific computing. *International Journal of High Performance Computing Applications*, 18:211–224, 2004.

[64] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 285–297, 1998.

[65] Magnus Karlsson, Fredrik Dahlgren, and Per Stenström. A prefetching technique for irregular accesses to linked data structures. pages 206–217, January 2000.

[66] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, 1994.

[67] Khronos. OpenCL: Open Computing Language. http://www.khronos.org/opencl, 2010.

[68] Induprakas Kodukula and Keshav Pingali. Data-centric transformations for locality enhancement. *Int. J. Parallel Program.*, 29(3):319–364, 2001.

[69] Kenneth Kundert. SPARSE 1.3. http://www.netlib.org/sparse.

[70] Los Alamos National Laboratory. Roadrunner – fact sheet. http://www.lanl.gov/news/newsbulletin/pdf/RRFactSheet.pdf, 2008.

[71] S. Langella, S. Hastings, S. Oster, T. Kurc, U. Catalyurek, and J. Saltz. A distributed data management middleware for data-driven application systems. In *CLUSTER '04: Proc. of the 2004 IEEE International Conference on Cluster Computing*, pages 267–276, Washington, DC, USA, 2004. IEEE Computer Society.

[72] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[73] Chris Lattner and Vikram Adve. Automatic pool allocation for disjoint data structures. In *MSP '02: Proc. of the 2002 workshop on memory system performance*, pages 13–24, 2002.

[74] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proc. of the international symposium on Code generation and optimization*, page 75, 2004.

[75] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, 2005.

[76] Chris Lattner and Vikram S. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proc. of the 2005 workshop on Memory system performance*, pages 24–35, 2005.

[77] D. Levinthal. Analyzing and resolving multi-core non scaling on Intel Core 2 processors. http://www.devx.com/go-parallel/Link/34762.

[78] Zhiyuan Li and Pen-Chung Yew. Program parallelization with interprocedural analysis. *Journal of Supercomputing*, 2(2):225–244, 1988.

[79] Yuan Lin and David Padua. Compiler analysis of irregular memory accesses. In *PLDI '00: Proc. of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 157–168, 2000.

[80] Andreas Löbel. MCF version 1.3 - a network simplex implementation. Available for academic use free of charge via WWW at www.zib.de, 2004.

[81] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-VII: Proc. of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, New York, NY, USA, 1996. ACM Press.

[82] B.M. Maker, R.M. Ferencz, and Hallquist J.O. NIKE3D: A nonlinear, implicit, three-dimensional finite element code for solid and structural mechanics - user's manual. (UCRL-MA-105268 Rev. 1), 1995.

[83] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14, 1991.

[84] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, 1960.

[85] Sjoerd Meijer, Hristo Nikolov, and Todor Stefanov. On compile-time evaluation of process partitioning transformations for kahn process networks. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 31–40, New York, NY, USA, 2009. ACM.

[86] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[87] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 140–152, 1988.

[88] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–18. USENIX Association, 2007.

[89] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proc. of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[90] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM.

[91] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 27, pages 542–555, 2008.

[92] NIST. Matrix Market. http://math.nist.gov/MatrixMarket, 2007.

[93] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[94] NVIDIA. CUDA: Compute unified device architecture. http://www.nvidia.com/object/cuda_home_new.html, 2010.

[95] NVIDIA. NVIDIAs next generation CUDA compute architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html, 2010.

[96] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *Int. J. Parallel Program.*, 27(3):131–159, 1999.

[97] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[98] Yale Patt. Requirements, bottlenecks, and good fortune: Agents for microprocessor evolution. *Proceedings of the IEEE*, 89(11):1553–1559, November 2001.

[99] B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Advances in Engineering Software*, 32(10-11):759–767, 2001.

[100] B. Patzák, D. Rypl, and Z. Bittnar. Parallel explicit finite element dynamics with nonlocal constitutive models. *Computers & Structures*, 79(26-28):2287–2297, 2001.

[101] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

[102] T. Quarles. SPICE3 version 3C1 users guide. Technical Report UCB/ERL M89/46, EECS Department, University of California, Berkeley, 1989.

[103] Shai Rubin, David Bernstein, and Michael Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *CC '99: Proc. of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 259–273, London, UK, 1999. Springer-Verlag.

[104] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31(4):251–283, 2003.

[105] Youcef Saad and Harry A. G. Wijshoff. SPARK: A benchmark package for sparse computations. In *ICS '90: Proc. of the 4th international conference on Supercomputing*, pages 239–253, New York, NY, USA, 1990. ACM.

[106] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5):603–612, 1991.

[107] H. R. Schwarz. *Finite Element Methods*. Academic Press, London, 1988.

[108] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *In SIGGRAPH 08: ACM SIGGRAPH 2008 papers*, pages 1–15. ACM, 2008.

[109] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. Design of Graphite and the polyhedral compilation package. In *Proc. of the GCC Developers' Summit*, pages 33–45, 2009.

[110] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[111] Greg Stitt, Gaurav Chaudhari, and James Coole. Traversal caches: a first step towards FPGA acceleration of pointer-based data structures. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 61–66, New York, NY, USA, 2008. ACM.

[112] Manuel Ujaldón, Emilio L. Zapata, Shamik D. Sharma, and Joel Saltz. Parallelization techniques for sparse matrix applications. *J. Parallel Distrib. Comput.*, 38(2):256–266, 1996.

[113] 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, 2008.

[114] Harmen L. A. van der Spek, Erwin M. Bakker, and Harry A. G. Wijshoff. Characterizing the performance penalties induced by irregular code using pointer structures and indirection arrays on the Intel Core 2 architecture. In *CF '09: Proc. of the 6th ACM conference on Computing frontiers*, pages 221–224, 2009.

[115] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. Automatic restructuring of linked data structures. In *LCPC '09: Proc. of the 22nd Int. Workshop on Languages and Compilers for Parallel Computing*, pages 263–277, 2009.

[116] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. How to unleash array optimizations on code using recursive data structures. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 275–284, 2010.

[117] Harmen L.A. van der Spek, Erwin M. Bakker, and Harry A.G. Wijshoff. Characterizing the performance penalties induced by irregular code using pointer structures and indirection arrays on the Intel Core 2 architecture. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 221–224, 2009.

[118] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007.

[119] Li Weng, Gagan Agrawal, Umit Catalyurek, Tahsin Kurc, Sivaramakrishnan Narayanan, and Joel Saltz. An approach for automatic data virtualization. In *HPDC '04: Proc. of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 24–33, Washington, DC, USA, 2004. IEEE Computer Society.

[120] Harry A. G. Wijshoff. Programming without bothering about data structures? *IEEE Comput. Sci. Eng.*, 3(3):67–68, 1996.

[121] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, 1989.

[122] Michael Wolfe. Loop skewing: the wavefront method revisited. *Int. J. Parallel Program.*, 15(4):279–293, 1986.

[123] Chia-Lin Yang and Alvin R. Lebeck. Push vs. pull: data movement for linked data structures. In *ICS '00: Proc. of the 14th international conference on Supercomputing*, pages 176–186, New York, NY, USA, 2000. ACM Press.

[124] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. February 1998.

[125] L. Zhao and H. A. G. Wijshoff. A case study in automatic data structure selection for optimizing sparse matrix computations. *Proc. of the IEEE International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems (IWACT)*, pages 22–55, July 2001.

[126] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 255–266, New York, NY, USA, 2004. ACM Press.

[127] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM, New York, NY, USA, 1991.

# Samenvatting

## Transparante Herstructurering van Datastructuren met Pointers

Processoren en computersystemen zijn zeer complex geworden. Een belangrijke factor hierin is simpelweg de steeds hoger wordende dichtheid van transistors (en dus ook het aantal) op een chip. Deze overvloed aan transistors werd in het verleden vooral aangewend om specifieke technieken te implementeren die de snelheid van een processor voor een enkele stroom van instructies verhoogt. Een voorbeeld hiervan is *pipelining*, een techniek waarbij de uitvoering van een instructie wordt opgedeeld in verschillende fasen, zodat er tegelijkertijd werk aan verschillende instructies kan worden verricht. Een ander bekend onderdeel van moderne processoren is de *branch predictor*. Deze voorspelt waar een branch instructie heen zal springen om zo weinig mogelijk vertraging te ondervinden van zulk soort instructies. Verreweg het grootste deel van de transistors wordt gebruikt voor caches, die veelgebruikte data uit het geheugen tijdelijk op de chip van de processor zelf opslaan. Het alsmaar sneller maken van één enkele processor stuitte op problemen, met name op het gebied van warmteontwikkeling. Dit heeft geleid tot de ontwikkeling van de huidige chips met meerdere processoren en andere hybride oplossingen.

Sinds het begin van het tijdperk van de moderne computer zijn veel datastructuren geïmplementeerd met behulp van pointers (verwijzingen naar geheugenlocaties). Het grote voordeel van datastructuren die met pointers zijn opgebouwd is hun flexibiliteit. Het invoegen van elementen is relatief eenvoudig en behoeft slechts aanpassing van enkele pointers. Echter, het gebruik van datastructuren die met pointers gebouwd zijn heeft als keerzijde dat de logisch structuur niet noodzakelijkerwijs overeenkomt met de fysieke structuur. Deze discrepantie zorgt ervoor dat de efficiency van algorithmen sterk afhankelijk is van de werkelijke layout in het geheugen. Een ander belangrijk probleem van datastructuren die met pointers gebouwd zijn is dat ze gebonden zijn

aan een specifieke adresruimte, wat migratie en parallellisatie van code en data op moderne parallelle systemen in de weg staat.

Dit proefschrift beschrijft en evalueert transformaties die het mogelijk maken om op pointers gebaseerde datastructuren te herordenen, zodat de fysieke layout van datastructuren aangepast kan worden tijdens de uitvoering van een programma. Door de layout aan te passen aan de manier waarop een datastructuur doorlopen wordt, kan de snelheid van een algorithme enorm toenemen. Een andere interessante bijkomstigheid is dat kennis van de layout, die bekend is nadat een datastructuur herordend is, gebruikt kan worden bij het optimaliseren van code.

Hoofdstuk 1 geeft een kort overzicht van de ontwikkelingen op het gebied van de computerarchitectuur en software voor parallelle systemen. De problemen die zich voordoen bij applicaties waar geheugentoegansgpatronen slecht voorspelbaar zijn worden beschreven in hoofdstuk 2. Bestaande technieken die betrekking hebben op deze problematiek worden besproken en de aanpak die verder uitgewerkt wordt in de rest van dit proefschrift wordt hier uiteengezet.

Om een juist beeld te krijgen van wat nu precies de impact is van onregelmatigheid en onvoorspelbaarheid van applicaties worden de SPARK00 benchmarks in hoofdstuk 3 gepresenteerd. Deze benchmarks worden gebruikt om verschillende eigenschappen van zulke applicaties te kwantificeren. Uit deze resultaten kan ook een referentiekader afgeleid worden, welke nuttig is om het effect van herstructurerende technieken te evalueren.

De herstructurerende technieken die beschreven worden in dit proefschrift worden in hoofdstuk 4 op een top-down manier uitgelegd. Aan de hand van voorbeelden in de programmeertaal C worden de concepten tastbaar gemaakt. In de volgende hoofdstukken worden deze concepten verder uitgewerkt. Hoofdstuk 5 bevat de benodigde achtergrondinformatie over het LLVM compilatiesysteem en beschrijft een bestaande compileranalyse die gebruikt wordt om datastructuren te identificeren die geschikt zijn om getransformeerd te worden.

De herstructurerende technieken om op pointer gebaseerde datastructuren te kunnen transformeren (en de technieken die nodig zijn om herstructurering mogelijk te maken) worden beschreven en geëvalueerd in hoofdstuk 6. Hierna worden deze technieken in hoofdstuk 7 verder uitgewerkt zodat code met pointers getransformeerd kunnen worden naar aan representatie die arrays gebruikt. Dit biedt ook de mogelijkheid om invariantie van *loopcondities* te monitoren, zodat *loops* eenvoudiger en beter geoptimaliseerd kunnen worden.

Hoofdstuk 8 beschrijft een aanpak om code in twee fasen te optimaliseren. Door de op arrays gebaseerde representatie te gebruiken zijn deze technieken in principe ook toepasbaar op applicaties die datastructuren met pointers gebruiken. De *sublimatietechniek* heeft als doel om code, die pointers en arrays met indirecte adressering gebruikt, te transformeren naar een tussenvorm welke geheel vrij is van pointers en indirecte adressering. Vervolgens wordt in de tweede fase de tussenvorm samen met specifieke data getransformeerd naar een voor specifieke data geoptimaliseerde code. In hoofdstuk 9 wordt deze techniek toepast om code die pointers gebruikt te transformeren naar een geoptimaliseerde code voor een FPGA die niet dezelfde adresruimte gebruikt als de *host CPU*.

Het optimaliseren van code is altijd al een uitdaging geweest en dit geldt in het bijzonder voor code waarin datastructuren die opgebouwd zijn met pointers worden gebruikt. In dit proefschrift worden praktische technieken voorgesteld die het potentieel hebben om deze categorie van moeilijk te optimaliseren en parallelliseren code ook automatisch te kunnen transformeren naar efficiëntere code. In de toekomst, waarin chips vele processoren zullen bevatten en hybride architecturen de norm zullen worden, zullen herstructurerende technieken van groot belang zijn om ook voor zulke code deze architecturen maximaal te kunnen benutten.

# Acknowledgements

I would like to thank all people who I have worked with during the past four years. In particular, I would like to thank my supervisor Harry Wijshoff, who has taught me the patience needed to conduct research. I also would like to thank Erwin Bakker, Mattias Holm and Kristiaan Rietveld, for all the interesting discussions we have had.

My parents have been of great support throughout the years and I greatly appreciate all the love they have shown. Finally, I would like to thank Erika, my wife, who encouraged me to be persistent in my research. Erika, thanks for sharing your life with me.

# Curriculum Vitae

Harmen Laurens Anne van der Spek werd geboren op 22 februari 1982, te Zevenhuizen (Zuid-Holland). In 2000 heeft hij aan het Driestar College te Gouda zijn VWO diploma behaald. Hierna is hij werkzaam geweest bij CS Engineering BV, te Waddinxveen, waar hij aan software werkte voor datacommunicatie en de beveiliging van netwerken. In 2001 is hij gestart met de opleiding Informatica aan de Universiteit Leiden. Gedurende de opleiding heeft hij zich breed geörienteerd, wat tot uiting is gekomen in het volgen van de vakken van de master track Bioinformatics. In 2006 heeft hij zijn doctoraal examen cum laude afgerond bij prof. dr. H.A.G. Wijshoff, waarna hij in dienst is getreden bij de Universiteit Leiden. Hier heeft hij samengewerkt met zijn promotor prof. dr. H.A.G. Wijshoff en copromotor dr. E.M. Bakker, en onderzoek gedaan naar nieuwe methoden om code- en datatransformaties uit te voeren op applicaties die datastructuren bevatten die met pointers zijn opgebouwd.