Optimizing pointer linked data structures

Holm, C.W.M.

Chapter 1

Introduction

Computers have come a long way since their beginnings with Charles Babbage's *analytical engine* in the mid 19th century and the introduction of electronic computers in the 1940s. In the last decades, a number of trends have been noticeable. One of the most fundamental trends has been the doubling of chip density every 18th month, a principle known as *Moore's law*. This doubling of chip density has led to the increase of processor performance at roughly the same rate, and although, lately, the sequential computing performance has not been increasing at the same rate as before, processors instead become more efficient at computing in parallel.

Another trend consists of the growing disparity between processor and memory speed. While processor speed has been increasing very fast, the speed of the memory that is used for storing programs and data has not been growing faster at the same rate. This disparity in processor and memory speed growth (known as the *memory wall* [58]), has for example led to problems in retrieving data from memory fast enough to keep up with processor speed. Computer engineers and scientists have devised several solutions to these problems, for example complementing computer systems with multi level high speed but lower capacity memory caches.

A cache is a small but fast memory that contains a copy of the data that is in main memory. The caches are filled automatically when the processor fetches data from memory so that additional accesses to the same location (or nearby) will go to the cache instead. The caches work well because data is regularly accessed multiple times within a short time interval, and data is often accessed nearby recently accessed data. This is known as temporal and spatial locality, respectively. Whenever the processor accesses data stored in

memory, the time it takes varies widely, depending on the data residing in the cache or not. Modern processors and memory systems can have access times over a hundred cycles[1], if the data accessed is not in the caches, while, if the data is in cache, the access time is a few or tens of cycles.

The more computer systems rely on hierarchically organized memory systems, the more it becomes critical for applications to have sufficient spatial and temporal locality. It is here where the main challenge lays for optimizing computations containing pointer linked data structures, as these computations include irregular access to memory. Caches that assume temporal and spatial locality, tend to have problems with irregular memory accesses, where data is accessed at more or less random locations. These irregular accesses tend to result in less than optimal utilization of hierarchically organized memory systems, especially when the problem size exceeds the cache size.

Several methods have been suggested in the past to assist with this situation, one of these being software based prefetching[8], where the program tells the processor which element to fetch next. Other techniques that may help are field elimination and reordering[20], where one optimize the data structure layout (in order to reduce its memory footprint). This thesis explores a novel way to improve temporal and spatial behavior of data. The technique explored is known as *data restructuring*, a method in which irregularly accessed data is reordered in memory into a more regular layout. We look at the problem from both the compiler, theory, programming language and hardware directions.

1.1 Compilers and Optimization

Compilers are programsthat translate human readable program code into machine code that is understandable by the processor. The codes compiled by the compiler consist of a number of fundamental parts: instructions embedded in control flow, and descriptions and definitions of data. The instructions and control flow describe how a problem is computed, while the data is the input, intermediate state and the output of a program.

It turns out that it is difficult to write human readable program code, and at the same time get the highest performance out of the program when it is executed. For this reason, the programmers and scientists who have been writing and developing compilers have developed methods that automatically generate more efficient machine code. These methods are commonly known as optimizations. A simple example that is fully automated is *instruction*

[1] A cycle is in simplified terms the minimum time in which a single instruction (e.g. adding two numbers) is executed.

assignment, where the best instruction for the job is selected, depending on the context. For example, a compiler will emit a shift operation (which takes one cycle) in the machine code, instead of a divide instruction (which takes over 20 cycles on modern processors[15]) when the divisor is a power of two. While, a programmer can easily replace divides with shifts in many cases, doing so will not only reduce the readability of the code and take time for the programmer, it may also differ in effectiveness, depending on the architecture of the machine. What is good to do on one machine may be bad on other machines.

Many of the optimizations that a compiler is able to do are a lot more complicated than the given example. Well known optimizations, such as for example strength reduction (where expensive operations are replaced with less costly ones), often take into account the control flow of the code, and others like loop unrolling and fusion (which reduce the relative cost of the loop counters compared to the loop bodies) take into account the memory cache architecture of the machine.

Compilers have so far been good at analyzing and optimizing what is known to the compiler at the moment of compilation, however for many, if not most programs, the data that is processed is not known at compile time. Therefore, the compiler must in many cases make assumptions on what the data will look like. If the assumptions are wrong, the optimized program may end up doing the wrong thing. As a consequence the assumptions made must be valid for the program under all circumstances and compilers adopt conservative approaches when it comes to analysis and optimization. With conservative approaches, compilers typically distinguish between what can be said to be true under all circumstances, what can be said to be false under all circumstances and what cannot be said to be true or false under all circumstances. For example, if a program has two pointers (references to an area of memory), the compiler may want to know whether they are the referring to the same memory area or not (this is known as *alias analysis*). A compiler would then by default, *conservatively* assume that they *maybe* are the same, before an analysis proves that the two pointers are always the same or never the same. Another example is dependency analysis, where the interesting property is whether or not two variables are independent. This means that the cases *proven dependent* and *unproven/unknown* are assumed to both mean that the variables are to be treated as dependent on each other.

1.2 Pointer-Linked Data Structures

A *pointer* is a fundamental concept of computing and programming. A pointer is a variable that contains an address of another variable. For more complicated programs, a common data type is a record which contains pointers to similar records. If several of these records are created, with pointers in the records storing the addresses of other records, we have what is known as a pointer linked data structure. Imagine that sets of records are chained using pointers, every part of a chain is a link, and from this we have the term *pointer-linked data structures*.

A pointer-linked data structure is a data structure which stores references to logically adjacent objects in pointers. One of the major advantages of pointer linked data structures is that they are dynamic in nature and can be constructed as the program runs. This is more difficult when using other data structures such as arrays, for which the size must be known before it is allocated and changing the size after allocation is expensive as the data must be moved.

There are two primary areas where pointer-linked data structures are commonly used: sparse data structures (where the the data structure may have implicit contents such as for example zeroes in the case of sparse matrices, avoiding storing the implicit content explicitly saves a lot of memory in many cases) and dynamic data structures that grow, shrink and change their shapes over time. For both types of data structures, the problem is irregularity, where objects that are accessed in sequence are not located adjacently in memory. These irregular data structures tend to make it difficult to exploit temporal and spatial locality, which means that the memory hierarchy cannot be used efficiently and the application will not live up to its best case performance.

While there exist different mechanisms to represent sparse and dynamic data structures in memory, this thesis focuses on the use of pointer-linked data structures for representing these structures in memory.

1.3 Data Restructuring

Data Restructuring is a method in which data in a program is transformed from one data layout to another. Examples of such transformations are the transformation of an *array of structs* (AOS) into *struct of arrays* (SOA), the reordering of data elements within an array, and the transformation of linked lists into arrays of structs.

There are primarily two types of restructuring possible, in the first one the

struct (or record) types are restructured, leading to for example the elimination of unused fields in a struct type or the reordering of the fields in a more optimal order. In the second form, the linked structs (also known as objects) are themselves reordered in memory in order to, for example, improve their regularity.

Compilers normally optimize the control flow of a program, however, with data restructuring, data that is not optimal for the processor may be reorganized in, for example, a more cache efficient way, or into forms that enable other optimizations such as for example parallelization and vectorization.

Note that, solving the exact layout for a data structure during runtime, *when the input is known*, is equivalent to solving whether the program (or parts of the program) will terminate or not. This is easy to understand as the program may always change the data structures just before a termination point. The problem is therefore *undecidable* (see [49] for explanation on the halting problem) and it is unfeasible to design a general algorithm that analyzes the exact shape of a program's data structures during its execution, even when the input is known. However, as the input for most practical applications is not known before runtime (input data is loaded from files known only at runtime), it is very difficult to analyze data layout at compile time and a general algorithm to find out the data layout of a program does not exist. Consequently, a system aiming to restructure data must be specific or conservative in its approach.

1.4 Outline

Data restructuring, can be done through a mixture of compiler analysis, runtime tracing and program transformations (as discussed in this thesis in Chapter 2), or manually by the programmer himself. However, other approaches are possible.

To this date, restructuring has to our knowledge been limited to data type transformations (e.g. structure splitting, field elimination and field reordering). For example, there were no attempts to carry out traversal pattern aware restructuring of pointer linked data structures.

Traversal patterns can be extracted in different ways, and in this thesis, Chapters 2, 4 and 5 take different approaches to this problem.

This thesis explores data restructuring from several viewpoints; from a pure compiler and runtime based approach, a graph theoretical approach, a programming language based approach, and a hardware based approach.

Chapter 2 describes the joint work on data restructuring carried out together with *Harmen L.A. van der Spek* as published in [54, 56]. In the chapter, we describe a new, generic restructuring framework for the optimization of data layout of pointer-linked data structures. Our techniques are based on two compiler techniques, pool allocation and structure splitting. By determining a type-safe subset of the data structures of the application, addressing can be done in a logical way (by pool, object identifier and field) instead of traditional pointers. This enables tracing and restructuring per data structure. We describe and evaluate our restructuring methodology, which involves compile-time analysis, run-time rewriting of memory regions and updating referring pointers on both the heap and the stack. Our experiments show that restructuring of pointer-linked data structures can significantly improve performance, while the overhead incurred by the tracing and rewriting is worth paying for.

Chapter 3 is based on joint work with *Hans L. Bodlaender* and goes into detail on the theory of grids, and especially sparse grids. Grids are very common data structures that appear in many codes. The chapter explores a graph theoretical approach suited for the analysis and restructuring of grid based pointer-linked data structures. The chapter introduces the new graph theoretic concepts of *confined components* and *strictly ordered orthogonality*. The Minimum Confined Components problem is shown to be NP-complete making the detection of arbitrary grids impractical without additional *a priori* known restrictions on the graph layout. However, for many applications, the theory introduced in this chapter can be applied successfully. Building on the concept of confined components, this chapter introduces the concept of strictly ordered orthogonality that is strongly related to grids. The notion enables the identification of rows and columns in an arbitrary grid graph, thereby enabling the formal transformation from pointer based data structures to array based data structures. A polynomial time algorithm capable of finding confined components is introduced and shown to be optimal for several types of graphs. In addition to this, the chapter introduces a polynomial time algorithm capable of verifying the strictly ordered orthogonality property. We show that for various grids, the underlying data structures of pointer traversing codes can be analyzed and potentially optimized in a feasible manner.

Chapter 4 introduces Pax C, a fully backwards compatible extension to the C programming language, that enables the programmer to specify restrictions on how pointer-linked data structures are constructed and used. Pax C consists of a set of programmer annotations and attributes (for example the ability to define how to reach all objects within a linked data structure) that

enable the compiler to automatically restructure data without resorting to (runtime) tracing information. The language extensions also contain a type qualifier that can be used to express static-pointer structures, pointer structures that have become constant at a certain point. We give the definitions of the attributes and show how we can apply the attributes on existing data structure definitions of some sample codes, including the Minimum Cost Flow program (MCF) from the SPEC benchmark series. The attributes entered by the programmer can be cross checked against the code either at compile or runtime, which allows for debugging of code. The language extensions are used to generate data restructuring transformations and the effects of these transformations are explored. We further show how the language extensions help the compiler exploit parallelism (in the MCF case, through minor modifications of the code) and turn pointer-linked data structures into position independent data, thereby potentially enabling automatic GPU optimization of the C-code.

Chapter 5 introduces a data restructuring method implementable in hardware. The method is based on a dedicated memory area that is used by the CPU to store pointer-linked objects next to each other. The chapter explores the system and includes a simulation-based approach comparing the overhead of the restructuring system to the performance gains achieved. The results show that the method is valid and performs at least as well as existing software-based approaches. A key contribution from this method is that, unlike previous software approaches, the hardware based approach is able to handle dynamically updated pointer structures in an automatic way and the method works without making any substantial modification to the programs.

Chapter 6 concludes the thesis and discusses future work.

1.5 Related Work

Chapters 2 and 4 in this thesis deal with the automatic or semi-automatic control of data layout in C. The C programming language is type-unsafe and in order to be able to automatically control the layout within type-unsafe languages, a type-safe subset of the program must be determined. If not, modifications on layout may have substantial effects on the result of the program. For example, if a program computes hashes on the binary contents of data structures (where pointers will convert into integers), the program is not type-safe. In this case, if an object is moved or has its fields reordered, the

result of the hash-function will change, thus if such a function is used, objects cannot be reordered since the program depends on the ability to convert pointer to integers and use them as such.

This section introduces some important related work, firstly the *Data Structure Analysis* algorithm capable of identifying the mentioned types-safe data structures. Secondly, we discuss a number of data restructuring transformations that depend on that algorithm. And, thirdly, we look into a number of analyses and different language extensions dealing with the determination of data structure layout, an important prerequisite for more advanced restructuring operations.

One algorithm to determine type safe subsets of C-programs is the *Data Structure Analysis* (DSA) algorithm. The DSA algorithm was developed by Lattner and Adve [35, 36]. The algorithm computes a data structure known as a *DSGraph* using a combination of local, bottom up and top down analysis of the call graph. The *DSGraph* describes how data structures are used at various points in the program, if a data structure is used in a type-unsafe manner, the *DSGraph* will indicate that the data structure is effectively an array of bytes, which implies that the data structure may have any kind of shape and should not be treated as type-safe. For type-safe pointer linked structures, the DSA will indicate whether or not two pointers points at disjoint data structures (as, these will have different nodes in the *DSGraph*). Note that the algorithm is conservative, and, therefore, there is only a guarantee that disjoint nodes represent disjoint objects in memory, but there is no guarantee that pointers referring to the same node represent the same object. The algorithm is described in more detail in Chapter 2.

The DSA has been used to enable different data restructuring methods. For example *automatic pool allocation* [35]. After the DSA algorithm has determined the type consistency and disjointness of pointer structures, the standard memory allocation primitives (*malloc*, etc.) are replaced with pool allocation primitives, where one pool is created per known disjoint object (lists nodes tend to end up as single objects in the *DSGraph*, so nodes in the same list, will be allocated from the same pool).

Two other transformations that use information provided by the DSA are *structure splitting* and *pointer compression*. Structure splitting is a transformation from an *array of structures* (AoS) into a *structure of arrays* (SoA). In *SoA* format the memory overhead of traversing pointer chains can be reduced as the *SoA* format will be free from padding and eliminate the loading of unused fields into the processor's memory cache. Structure splitting has been implemented by several researchers [12, 20, 54, 22]. In [12, 54] descriptions are given on how the DSA assisted pool allocation can be used to automati-

cally split the pool allocated structures. The former paper describes a system implemented for the IBM XL Compiler and the latter a system using the LLVM backend compiler. Hagog and Tice have implemented a similar method in GCC [22] (but without using DSA). The GCC-based implementation does not seem to provide the same information as DSA. Strictly taken, structure splitting is not necessary for dynamic remapping of pointer structures, but it simplifies tasks like restructuring and relocation considerably. Moreover, splitting simply has performance benefits because data from unused fields will not pollute the cache.

The DSA which is a points-to-analysis (it determines whether pointers point to different objects), should not be confused with *shape analysis*. Shape analysis concerns the shape (e.g. tree, DAG or cyclic graph) of pointer-linked data structures. Ghiya and Hendren proposed a pointer analysis that classifies heap directed pointers as a tree, a DAG or a cyclic graph [19].

Hwang and Saltz realized that it is of more importance how data structures are actually traversed instead of knowing the exact layout of a data structure. They integrated this idea in what they call *traversal-pattern-sensitive* shape analysis [27].

Graph Types [30] and *PALE* [41] introduced languages that allowed for properties on a pointer-linked data structure to be proven. It was possible to prove and disprove that code obeyed certain properties (such as, for example, that a list tail pointer actually pointed to the tail of the list). While being powerful in the proof mechanism, the PALE system did have a high space and time complexity. And the examples that were used to evaluate PALE were relatively small in size. If the measurements of the PALE experiments were extrapolated linearly for larger projects (million lines of code), memory usage would quickly approach terabytes of memory and execution times would be measured in days. It should be said that PALE is interesting for model checking of small kernels for safety critical systems.

Notable in this area is also the *Abstract Description of Data Structures* (ADDS) [24] and its generalization *Abstract Specification of Aliasing Properties* (ASAP) [26]; ADDS extends the type syntax, allowing the programmer to associate pointers in a data structure with named dimensions in an overlaying data structure. The ASAP generalization instead allows the use of path expressions to declare disjointness of paths in the data structure. These approaches both add powerful properties to the data structure description allowing the compiler in turn to determine the aliasing of two adjacent objects in for example a pointer chasing loop. Both are type description languages suitable for describing alias properties, but they were not integrated in an existing programming language like C.

Another project known as *shape types* [17], did add non-backwards compatible extensions to the C language. In shape types, shape restrictions on data structures where added using a context free grammar based syntax, the purpose being similar to PALE. One of the major problems with *shape types* is its rather complicated syntax and semantics.

A Note on Units

In this thesis, data sizes are given in units compliant with existing standards. This means that the symbol for byte is a capital *B*, the symbol for bit is either a small *b* or *bit*. Prefixes have their meaning as specified in the SI system. For example: *GB* is equal to one billion (short scale) bytes. In the cases where powers of twos are needed, as with memory sizes, the binary prefixes as defined by *ISO/IEC 80000* are used. Consequently a *KiB* is 1024 bytes and a *GiB* is 1073741824 bytes.