Optimizing Pointer Linked Data Structures

Proefschrift

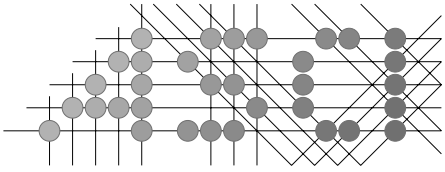
ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op donderdag, 31 januari, 2013
klokke 16:15 uur

door

Carl Wilhelm Mattias Holm
geboren te Täby, Zweden in 1980

Promotiecommissie:

Promotor:	Prof. Dr. H.A.G. Wijshoff
Overige leden:	Prof. Dr. K.A. Gallivan (Florida State University)
	Prof. Dr. Ir. E.F.A. Deprettere
	Prof. Dr. F.J. Peters
	Dr. E.M. Bakker
	Prof. Dr. J.N. Kok



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 268.

Optimizing Pointer Linked Data Structures
Carl Wilhelm Mattias Holm
PhD Thesis, Universiteit Leiden

ISBN: 978-90-8891-552-9
Printed by: Proefschriftmaken.nl
Published by: Uitgeverij BOXPress, Oisterwijk

To Adinda

Contents

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.

Chapter 2

Pointer Structure Restructuring

Predictability in memory reference sequences is a key requirement for obtaining high performance on applications using pointer-linked data structures. This often goes against the dynamic nature of such data structures, as pointer-linked data structures are often used to represent data that dynamically changes over time, which will reduce the predictability, even if the pointer-linked structure was in perfect order initially. Also, different traversal orders of data structures cause radical differences in memory reference behavior when considering the data layout.

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

This chapter is based on earlier work done with *Harmen van der Spek*, so parts of this chapter also appeared in his thesis [50].

domain specific memory allocation and rewriting of data structures.

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), a conservative whole-program analysis reporting on the usage of data structures in applications [35, 36]. The analysis results 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, i.e. they store only data of a specific (structured) type. These pools are our starting point.

For type-homogeneous pools, we have implemented *structure splitting*, similar to MPADS [12], 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. It is important to note that tracing and restructuring all happen within *a single run* of an application.

In order to illustrate the need for restructuring, it is interesting to have a look at what could potentially be achieved by controlling data layout. For this, we used SPARK00 [55, 52], a benchmark set in which the initial data layout can be explicitly controlled. Figure 2.1 shows the potential speedups on an Intel Core 2 system (which is also used in the other experiments, together with its successor, the Core i7) if the data layout is such that the pointer traversals

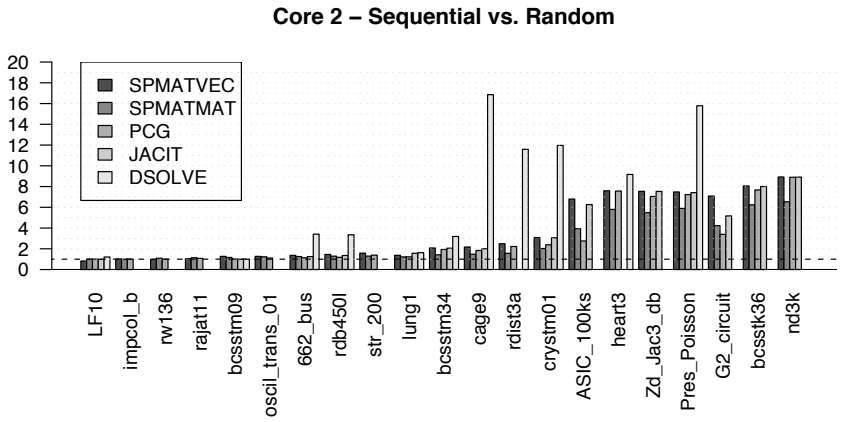


Figure 2.1: Speedup on SPMATVEC when using data layout with sequential memory access vs. layout with random memory access on the Intel Core 2 architecture.

result in a sequential traversal of the main memory, compared to a layout that results in random memory references. This figure illustrates the potential for performance improvements if data layout could be optimized. Our framework intends to exploit this potential for performance improvements.

Section 2.1 starts with an explanation of work on Data Structure Analysis, that our restructuring framework depends on. Section 2.2 describes the compile-time parts of our framework, while Section 2.3 treats the run-time components. Section 2.4 contains the experimental evaluation of our framework. Restructuring pointer-linked data structures has great potential and in this chapter 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 tracing mechanism, which of course does not come for free, is discussed in Section 2.4.2. 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 2.4.3. Address calculations need to

be efficient. Therefore, we present improved address calculations, compared to the address calculations in Curial's work [12], for addressing split memory pools in Section 2.4.4. Related work is discussed in Section 2.5. Future work and conclusions are given in Section 2.6. Part of this chapter has appeared in [54].

2.1 Preliminaries

The restructuring framework presented in this chapter relies on the fact that a type-safe subset of the program has been identified. This is achieved by applying Lattner and Adve's Data Structure Analysis (DSA) [34, 38, 35, 36]. DSA is an efficient, inter-procedural (whole program), context- and field-sensitive pointer analysis. It is able to identify (conservatively) 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. We will not describe how DSA works in detail, but we will explain the meaning of the resulting Data Structure Graph (DS Graph) as this forms the basis for our further analyses and transformations.

For implementation and efficiency reasons, data structures are not stored as they have been defined in the original source code. As the DSA provides us with 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 as we know it (otherwise it would not be type-safe).

The pool restructuring framework that we propose in this chapter is based on two techniques: automatic pool allocation and structure splitting. The structure splitting transformations remaps memory pools of records into structured data that is grouped by field instead (essentially, it is mapping from an array of records to a record of arrays). The implementation developed is similar to the MPADS framework of Curial et al. [12], though we optimized the address calculations for commonly occurring structure layouts (Section 2.4.4).

In this section, both DSA and structure splitting, which our analysis passes and transformations depend on, are explained in further detail.

2.1.1 Data Structure Analysis

Data Structure Analysis (DSA) provides information on the way data structures are actually used in a program. First, it is important to understand that

```
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 2.2: Code excerpt of main function of SPMATVEC.

DSA is *not a shape analysis*. DSA determines which data structures can be proved 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 [34]:

> 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 (all memory objects represented by the node are of the same type and are used in a type-consistent way throughout the entire program.

Construction of the DS graph occurs 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. DS nodes contain flags that indicate whether they contain complete information. The subsequent phase, the Bottom-Up Analysis Phase, combines the information on the local functions with results from their callees, by propagating this information bottom-up. This phase is context-sensitive. The last phase is the Top-Down Analysis phase, which we will not need in our restructuring framework. We use the result from the Bottom-Up Analysis.

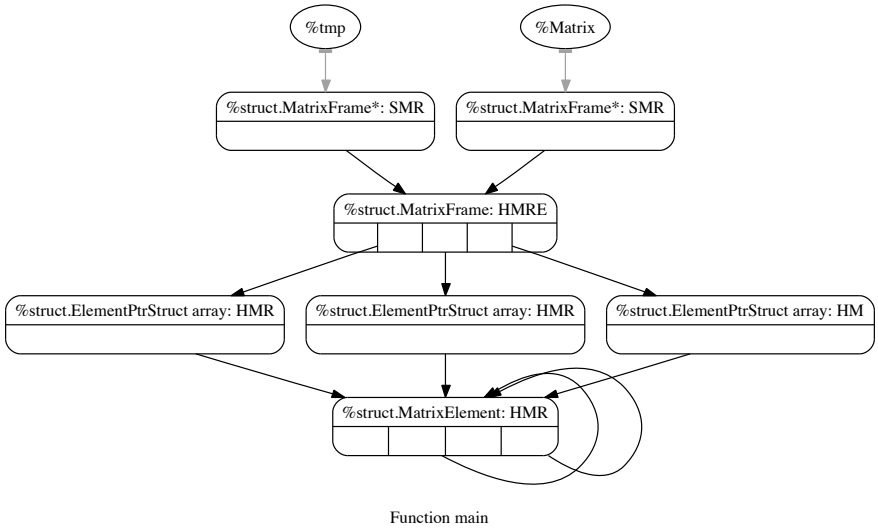


Figure 2.3: DSGraph for main function of SPMATVEC benchmark.

Let us illustrate this with an example. Figure 2.2 shows a part of the main function of SPMATVEC, one of the benchmarks used in the evaluation of our method (see Section 2.4, the full sources are available online [51]). Figure 2.3 shows the associated DS Graph. 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 $\%Matrix$. Each of these variables has its own 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, that 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 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 information is propagated to the point where the data structure is instantiated. At that location, a choice can be made about how this data structure instance is treated.

Summarizing, for each data structure, we are interested in the point at which 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.

2.1.2 Automatic Pool Allocation

On top of DSA, Lattner et al. implemented *automatic pool allocation* [35, 36]. Pool allocation is a transformation that replaces calls to memory allocation functions by 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. After a node in the DS Graph has been determined to be type-safe, all associated memory allocation functions can be identified and be rewritten such that they call a pool allocation library, whose functions take an additional argument, the *pool descriptor*, that uniquely identifies a data structure instance at run-time. Pool allocated structures allow for precise control on data layout, as it is known that all allocated elements within a particular region have the same type. We use this property to modify the way structure are laid out in main memory.

2.1.3 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. Such a pool is just an array of structures (AOS). If we assume that the size of this array is fixed, the AOS can be easily transformed into a structure of arrays. Figure 2.4 depicts this concept by giving the corresponding structure definitions in C.

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) because primitive data types (i.e. floats, doubles, integers etc..) normally must be aligned to addresses corresponding to 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 the many cases where padding is normally

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

Figure 2.4: Array of structures vs. structure of arrays.

inserted. 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 because in the split version multiple fields will be located in 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. [12], who implemented structure splitting in the IBM XL compiler.

2.2 Compile-time Analysis and Transformation

At compile-time, a whole program transformation is applied in order to rewrite pools to use a split structure layout that supports run-time restructuring. Figure 2.5 shows an overview of the entire compilation chain for our framework. In this section, the analyses and rewriting compiler passes are discussed.

2.2.1 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 then start at the *main* function and traverse all reachable

[1] Translation Look-aside Buffer

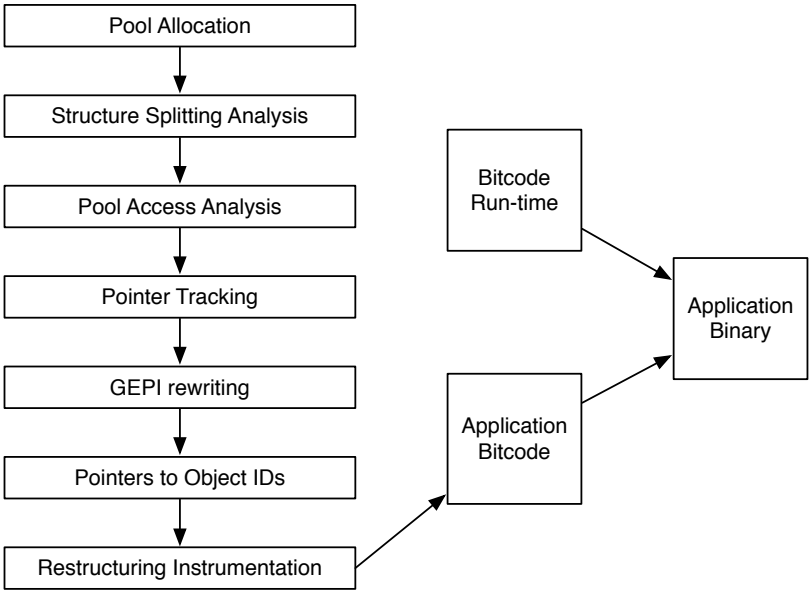


Figure 2.5: Overview of the pool restructuring compilation chain. GEPI refers to the LLVM GetElementPtrInst instruction.

functions, cloning each function that needs to be rewritten to support the data layout of split structures. Note that cloning is only done along execution paths that are known to have type-safe data structures that can be split safely. Functions are cloned because there might also be calling contexts in which splitting cannot be applied, and these cases must also be dealt with correctly (see Lattner and Adve's work [35, 36]). From this work, we also use the their technique for the identification of the memory pools. It is not possible to split pools that are not type homogeneous because addressing of object fields would become ambiguous and fields of different types and length would introduce aliasing of field values. This information is 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, because 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 to be 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. These loads are needed to support the use of *object identifiers* instead of pointers (see Section 2.2.5). 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. Instead, they are rewritten just before code generation because additional passes will need to reason about these expressions.

2.2.2 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 runtime, *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 more precisely. Lattner and Adve's pointer compression applies the same restriction on field accesses [37].

2.2.3 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 mean 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

Method	Advantages	Disadvantages
Pointer Tracking	Simple Portable	Slow Interferes with IR
Shadow Stack	Fast Portable	Interferes with IR
Stack Map	Fast No IR Interference	Backend Modifications Stack walking not portable

Table 2.1: The three stack management options and their individual advantages and drawbacks

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 2.1.

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 in this case also needed to enable the removal of all the pointer tracking entries associated with a returning function. In LLVM this means that any pointer that is an SSA register must explicitly stored on the stack. The following LLVM function illustrates this a bit further:

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

The function listed above is 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 named *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. These run-time functions are very short (a few instructions) and will be inlined and thus, do not induce any function calling-overhead. Figure 2.6 shows how the pointer tracking block is constructed during run-time.

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:

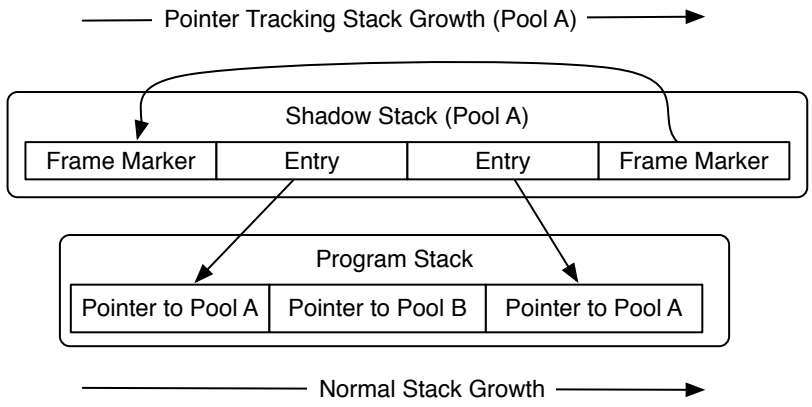


Figure 2.6: Pointer Tracking Layout

```
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.

The most important point with the explicit pointer tracking, is that it took a small implementation effort compared to the methods described further on in Sections 2.2.3 and 2.2.3. So, while the method (as shown in experiments later on) is not a good choice for a fielded 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 [23]. To implement shadow stacks the compiler creates a per function data structure where pointers that are stored on the stack will be stored as a group, such that each pointer can be addressed relative to the base of this data structure. When a function is called, such a structure is allocated on the stack and this structure is then registered with the runtime. This pre-registration cuts down on the additional registration overhead compared to the pointer tracking, by only inferring one registered pointer per function call.

Stack Maps

The third alternative is the construction of stack maps (for example described by Agesen in [1]). 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 about, 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.

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 no longer have constant offsets from the base pointer of the structure (see Figure 2.7 for a graphical explanation of why this happens).

2.2.4 Address Calculations

It is obvious that calculating addresses for the fields in the structures must be very efficient. This fact was already stressed by Curial [12], 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:

$$offset_n = k_n + sizeof_n \frac{p\&(sizeof_{pool} - 1)}{sizeof_0} - p\&(sizeof_{pool} - 1) \tag{2.1}$$

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[2].

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:

$$offset_n = k_n + (sizeof_n - sizeof_0) \frac{p\&(sizeof_{pool} - 1)}{sizeof_0} \tag{2.2}$$

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

$$offset_n = k_n - (sizeof_0 - sizeof_n) \frac{p\&(sizeof_{pool} - 1)}{sizeof_0} \tag{2.3}$$

Equation 2.1 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 2.7. This figure also demonstrates that Equation 2.2 and 2.3 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 divides involving a power of two constant. Fog [15] 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 neither $sizeof_0$ nor $sizeof_n$ are powers of two, Equation 2.1 will take 26 cycles. If $sizeof_0$ is a power of two the same equation will take 6 cycles (as

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

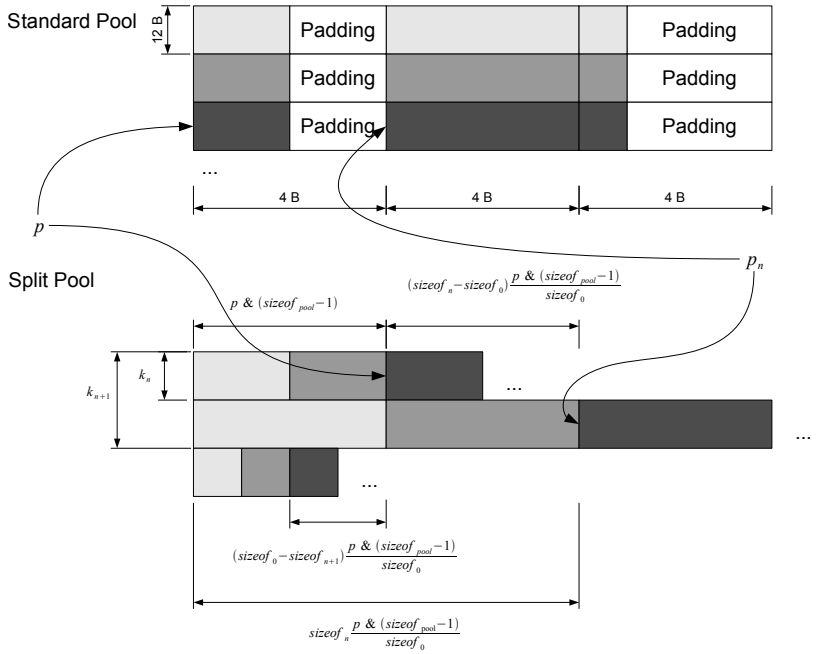


Figure 2.7: Graphical representation of split pools and the field offset expressions detailed in Section 2.2.4 for a split pool consisting of structures with elements of sizes 2, 4 and 1 bytes. Each shade of gray represents an individual object. The object pointer p is in this case is pointing at the third object and the derived pointer p_n is pointing to the second field of the third object.

the very costly divide will be reduced to a shift) and if both sizes are powers of two it will take 4 cycles. Equation 2.3 will take 3 cycles, and Equation 2.2 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 2.1 and the elimination of calculations if accessing the first field are already used in MPADS [12], but our additional Equations 2.2 and 2.3, 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 and that the third term in Equation 2.1 has been eliminated. Note that for Equation 2.2 when $size_n - size_0 = size_0$,

LLVM will automatically eliminate the multiply and the divide instruction, giving even more savings.

The most notable equation cost (26 cycles) come from the existence of a divide instruction in the expression. 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 because the divide instruction has been eliminated through strength reduction introduced by Equation 2.3.

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. Though, at this moment our implementation does not do this and this field reordering remains on the future work list.

2.2.5 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 2.2.2, 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 Latttner and Adve's *static pointer compression* [37]. 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 object 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 because the representation is position-independent. Whenever a pointer is loaded from memory, the conversion is done in an architecture and context-dependent way.

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

Figure 2.8: Store Value Rewrites

Section 2.2.2 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), the pointer value that will be stored into the memory pool needs to be converted to an object identifier before it is stored. When such a pointer value is loaded from a memory pool, it must be converted from an object identifier to a pointer. Loads and stores to the stack are unaffected and thus will contain real pointers. As no pointers to fields, but only pointers to objects will be stored to the memory pools, we only need conversion functions for object pointers. For store instructions, the value to store is rewritten as illustrated in Figure 2.8 and for load instructions, the loaded value is rewritten as illustrated in Figure 2.9

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 [37], 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

```
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 *)obj;
  }
}
```

Figure 2.9: Load Value Rewrites

deal with full pointers.

2.2.6 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 and 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.

We only enable tracing for one execution of a function and its callees because tracing is a method that does not come for free. After this first tracing, 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 that is set to the non-traced version after a trace has been obtained.

2.3 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 provided for regular pool allocation. The split-pool runtime provides *create* and *destroy* functions to split pools and for 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.

2.3.1 Application Programming Interface

The split-pool run-time offers implementations for initializing pools and for memory allocation and deallocation. Tables 2.2 and 2.3 describes the run-time functions needed to support restructuring of split pools.

2.3.2 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 2.2.2) provides the compile-time information (pool, object and field) about all memory references and these memory references can all be traced. Traces are generated per pool, and per field. 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 object identifiers encountered 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++;
  }
}
```

Function	**Arguments**	**Return value**	**Description**
split_pool_vargs	• split_pooldesc_t *pool • uintptr_t obj_cnt • uint32_t unsplit_obj_len • uint32_t split_obj_len • uint32_t field_cnt • ...	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 • 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*.

Table 2.2: Functions of the split pool run-time with restructuring support.

Function	Arguments	Return value	Description
split_poolrealloc	• split_pooldesc_t *pool • void *obj • unsigned NumBytes	void *	**Reallocate Memory from a Split Pool.** Replacement for *realloc* in the standard C library.
split_poolfree	• split_pooldesc_t *pool • 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 split_pooltrace_trace_base_heap	• void *pool • uint32_t field • 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 2.3: Functions of the split pool run-time with restructuring support.

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

Tracing does not come for free and, therefore, tracing should be avoided if it is not necessary. For the evaluation of our restructuring method we choose to trace the first execution of a specified function (compiler option specifies which function), restructure using this trace and then disable tracing. 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).

2.3.3 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 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 2.3.2. 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.

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:

```
for (referrer in pool.referrers) {
  for (ent in referrer.field) {
    referrer.field[ent] = permVec[referrer.field[ent]];
  }
}
```

Here, each pool that refers to the rewritten pool will have the field containing those pointers updated with the new locations.

The algorithm detailed here assumes that each pool descriptor has information available regarding the pool connectivity (i.e. which fields in other pools that points out 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.

2.3.4 Stack Rewriting

As already discussed in Section 2.2.3, 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 to which it refers.

2.4 Experiments

The challenge of a restructuring compiler is to generate code that will automatically restructure data, either at compile-time or run-time, in order to achieve 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. In the experiments here, we ideally want to obtain similar performance gains, but by automatic restructuring of data layout of the used pointer-linked data structures.

We use the benchmark set SPARK00 which contains pointer benchmarks whose layout can be controlled precisely [55, 52]. 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-modification 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

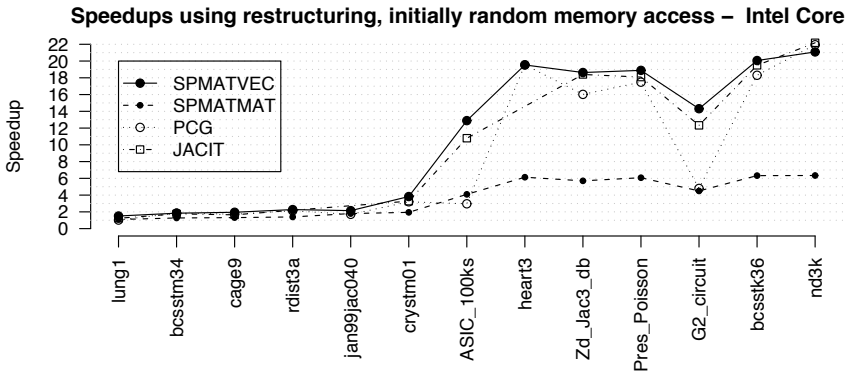
based system with 6 GiB of main memory, running Ubuntu 9.04.

2.4.1 Pool Reordering

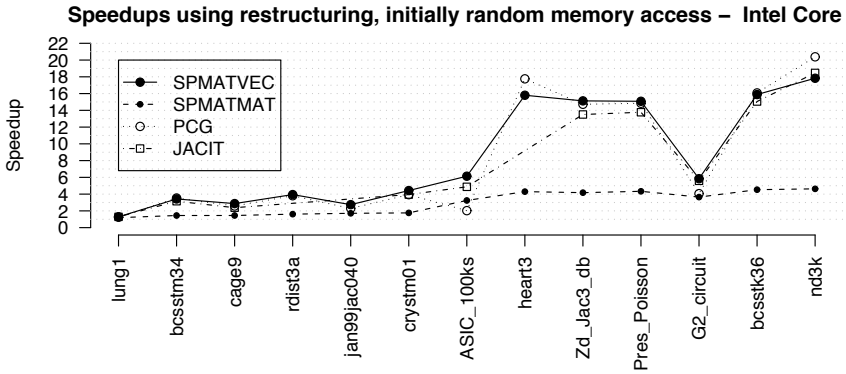
As shown in the introduction, being able to switch to an alternative data layout can be very beneficial. We applied our restructuring transformations to the SPARK00 benchmarks and show that in ideal cases, speedups exceeding 20 are possible by regularizing memory reference streams in combination with 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 2.4.2 to allow a better comparison between the different data sets.

As a first step in our experimentation, we first determine the maximal improvement possible, by using an initial layout that causes random memory access. Figure 2.10(a) and 2.10(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 [55, 52], 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 a 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 2.11(a) and 2.11(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, which 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

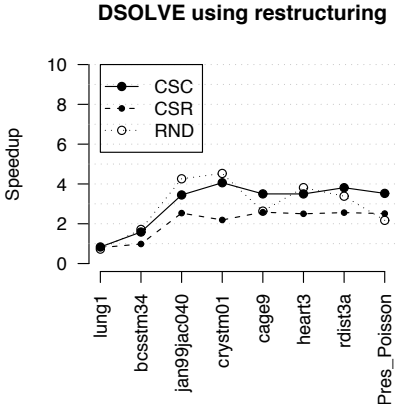


(a) Intel Core 2

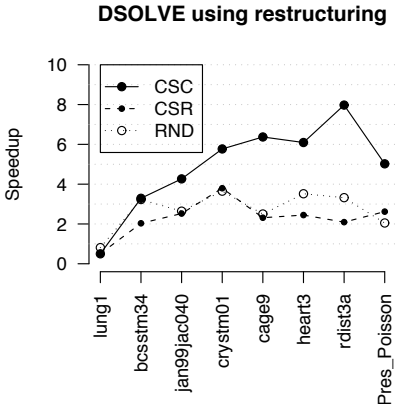


(b) Intel Core i7

Figure 2.10: Speedups obtained using restructuring on the SPARK00 benchmarks. The initial data layout is random.



(a) Intel Core 2



(b) Intel Core i7

Figure 2.11: Speedups obtained using restructuring on DSOLVE for all different initial layouts. Input data sets are ordered by size (after LU-factorization).

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× 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 has been 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 2.3.2). Generation of permutation vectors from traces will be improved in future versions of the framework.

2.4.2 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 an appropriate trace. This will be addressed in a forthcoming paper.

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 2.3.2. So for instance, if a linked list contains a floating point field and this list is summed using a list traversal, then if 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. 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, using a data layout causing random memory access. Figure 2.12 shows 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

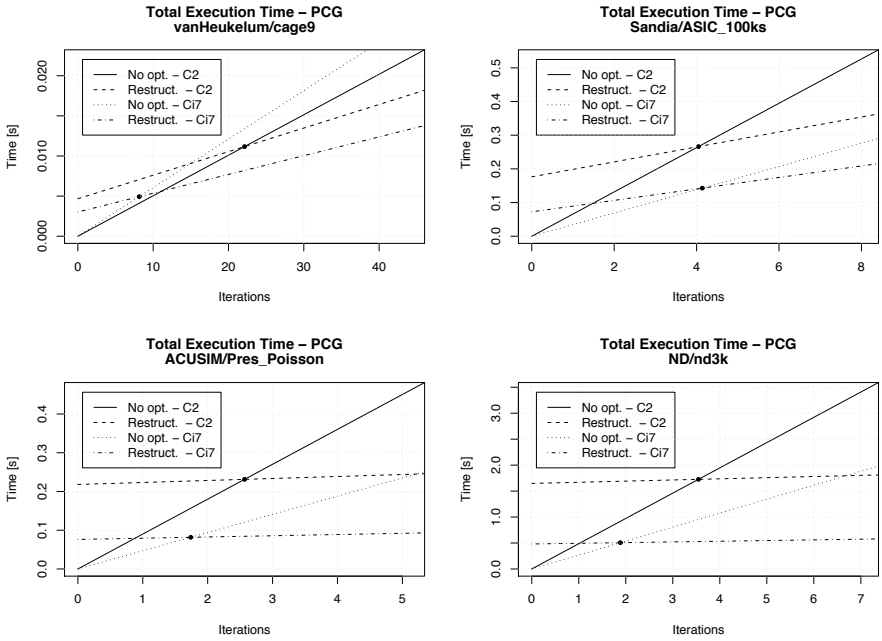


Figure 2.12: Execution times with and without restructuring. The break-even points are marked with a dot.

	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 2.4: Number of iterations for the break-even points when tracing and restructuring is enabled, and 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.

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 2.10(a) and 2.10(b)). The break-even points for all matrices are included in Table 2.4.

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 2.12, 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 on this architecture.

Although we have shown in this section that the additional costs of tracing are manageable, it should be noted that we only showed this on computational kernels. In general, it is not recommended to trace a full application code. Therefore, as we have noted earlier in this chapter, tracing should be turned on explicitly by a compiler option and coupled with a specification of the functions that should be traced.

2.4.3 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 overhead in this case will be a measurement of per-function and per-pointer overhead.

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 in term 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 unmanaged version should thus represent the overhead introduced for that number of pointers in the given number of calls to the function.

Figure 2.13 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).

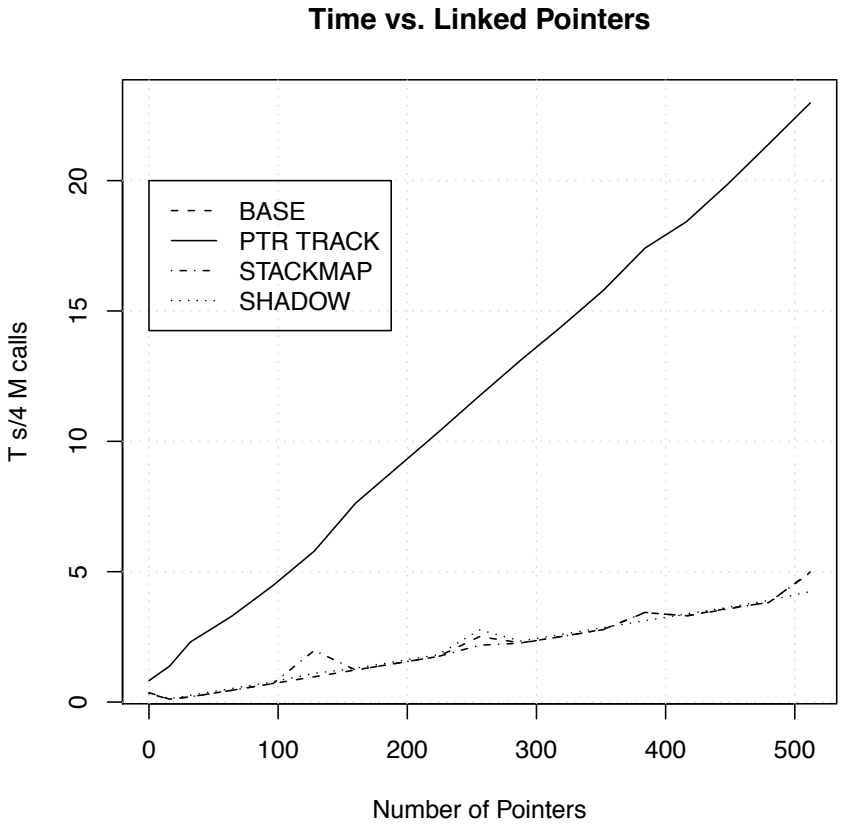


Figure 2.13: Execution time of a function with different stack management approaches

Since the shadow stack and stack map strategies have not been implemented and thus these strategies have not been evaluated using compiler generated code, a hand-written implementation of these strategies has been used to estimate the overhead of these techniques.

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 needs to be registered. This has its own problems, as it prevents certain optimizations 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.

2.4.4 Address Calculations

The address calculation expressions used are an improved variant of those introduced by Curial et.al. [12]. These improvements have been verified experimentally by running two versions of the pointer-based applications from the SPARK00 benchmark suite [55, 52], one with the new optimized address calculation expressions enabled, and one version with only the general addressing equations used by MPADS enabled. It should be noted that the implementation described in this chapter 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 in this chapter has not been carried out.

The matrix input files are sparse and inserted in row-wise order, leading to a regular access pattern upon traversal. In Figure 2.14, 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*-linker and the pool allocation 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 or may not do due to the changed instruction stream.

Table 2.5 gives the average improvements of the addressing optimizations. In Table 2.5, the SPMATVEC benchmark actually lost in performance, this was due to instruction cache conflicts in the new code. Figure 2.14 shows the general behavior of the benchmarks where the relative performance improvements is greater for smaller data set sizes. This is because the new instruction

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 2.5: Performance gain averages 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

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 overall influence.

2.5 Related Work

Optimization of data access in order to improve performance of data-intensive applications has been applied extensively, either by automatic transformations or by hand-tuning applications for efficient memory access. In some cases, memory access patterns can be determined symbolically at compile-time. In such cases, the traditional transformations such as loop unrolling, loop fusion or fission and loop tiling can be applied. For applications using pointer-linked data structures, such techniques can in general not be applied.

The traditional methods mentioned above change the order of instruction execution such that data is accessed in a different way, without affecting the result. One might as well change the underlying data layout, without affecting the computations. This is exactly what has been done on pointer-linked data structures in this chapter.

In order to be able to automatically control the layout within type-unsafe languages such as C, a type-safe subset must be determined. The Data Structure Analysis (DSA) developed by Lattner and Adve does exactly that [35, 36]. It determines how data structures are used within an application. This has been discussed in Section 2.1.1.

DSA 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

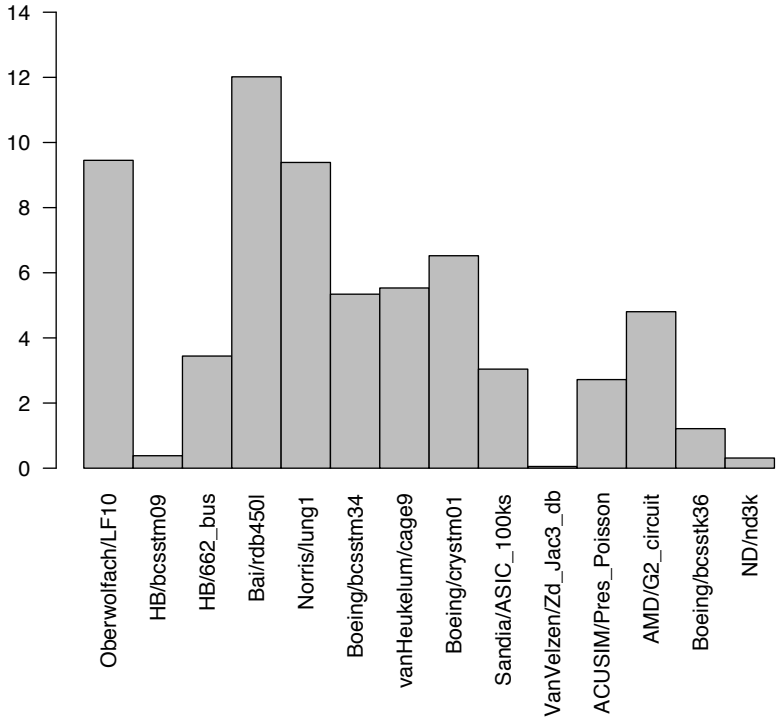


Figure 2.14: Typical speedup in percent (%) (in this case for the *JACIT* benchmark) ordered by increasing matrix size, from left to right.

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]. Integrating such an approach in our compiler could help in reducing the overhead introduced by the pool access tracing by traversing data structures autonomously in the

run-time.

Type-safety is essential for data restructuring techniques. Two other transformations that use information provided by the DSA are structure splitting and pointer compression. Curial et al. implemented structure splitting in the IBM XL compiler, based on the analysis information provided by the DSA [12]. Hagog and Tice have implemented a similar method in GCC [22]. 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.

Data layout optimization can also be provided by libraries. Bender and Hu proposed an *adaptive packed-memory array*, which is a sparse array that allows for efficient insertion and deletion of elements while preserving locality [4]. Rubin et al. take a similar approach by grouping adjacent linked-list nodes such that they are colocated in the same cache line. They call this approach *virtual cache lines* (VCL) [43]. They state that they believe that compilers will be able to generate VCL-based code. We believe our pool restructuring achieves this automatic remapping on cache lines (albeit in a different way). In addition, cache usage is very efficient after restructuring a memory pool because our implementation employs full structure splitting,

Rus et al. implemented their Hybrid Analysis that integrates static and run-time analysis of memory references [44]. Eventually, such an approach might be useful in conjunction with our restructuring framework to describe access patterns of pointer traversals. Saltz et al. describe the run-time parallelization and scheduling of loops, which is an inspector/executor approach [45]. Our tracing mechanism is similar to this approach, as it inspects and then restructures. The future challenge will be to extend the system such that it inspects, restructures and parallelizes.

2.6 Conclusions

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, and 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 [12]. For static analysis, this may often be true. For

dynamic analysis, 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 when data structures are used repetitively.

The restructuring framework described in this chapter opens up more optimization opportunities that we have not explored yet. For example, after data restructuring extra information on the data layout is available and could be exploited in order 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 [37]. However, with the pool restructuring presented in this chapter, a more detailed segmentation of the pools can be made and restructuring could be extended to a distributed pool restructuring framework.

The implementation presented 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 described 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 3

Theory of Grids

Data structure transformations still need a substantial research effort to make them more effective. In this chapter, possible future directions for empowering data structure transformations are explored. Although not directly implementable, the techniques and methods may very well be so in the future when further research has been conducted in this area. The feasibility of restructuring data structures is explored by treating the restructuring and analysis of the linked data structures as a graph problem. In essence we try to redefine the topology of the data structure used in such a program into a more optimal data structure, given the actual structure of the graph. For example, if a graph representing the data structure has a Hamiltonian path, it is possible to restructure that part of the data structure to an array.

In general, we will not consider control flow in the analysis, as for instance is done in the work on shape analysis [19], but only analysis of the topology of the graph. As such this is the first step in handling the restructuring problem and the methods need to be extended with control flow analysis in the future. An important result in this chapter is that we show that control flow analysis, or other additional knowledge, is required in order to effectively identify sparse grids and other regular structures, although it is feasible to apply direct approaches on complete grids. This result is the guiding principle that led to the later chapters in this thesis.

The shape of a pointer linked data structure cannot be determined from type information only (pointers point to arbitrary objects). For example, a structure type containing two recursively typed pointers in each object, may represent a doubly linked list, a tree, a grid, or any other kind of graph formable by two outgoing arcs per vertex (and by edge decomposition, many other kinds

of graphs may be formed, including complete graphs).

Special optimizations may be carried out on for example lists, trees and grids in some cases. For example, a list is linearizable (i.e. can be transformed into an array), grids are two dimensional and can be transformed into two dimensional arrays (using for example a jagged diagonal layout). In order to carry out a linearization of a grid it must be known by the compiler or the run time that the structure actually is a grid and not for example a doubly linked list or a tree.

In these cases it is important that there exists a theoretical base in order to choose the right methods when doing such optimizations or related analysis.

This chapter introduces the problem named MINIMUM CONFINED COMPONENTS, which is related to finding *rows* or *columns* in graphs. MINIMUM CONFINED COMPONENTS is shown to be NP-complete. However, for special classes of graphs such as complete grids and trees, the MINIMUM CONFINED COMPONENTS problem can be solved in polynomial time, and for many other graphs, we can use a heuristic method for decomposing a graph into confined components. There are several areas where confined components have both theoretical and practical applications. We give an algorithm capable of decomposing a graph into confined components in $O\left(|V|^2 + |V||E|\right)$ time. This algorithm is shown to be optimal for trees, complete grids and complete triangular grids.

We introduce the concept of orthogonal sets of confined components that can be used to formally describe grid styled graphs. For many cases it turns out that we do not need to compute the orthogonal sets but can instead make a good guess that can be proven to be orthogonal using a simple but efficient algorithm. We introduce such an algorithm that is also mostly parallel, with a sequential complexity of $O\left(L * (|V| + |E|)\right)$, where L is the number of dimensions in the grid.

Several practical applications are discussed in detail, giving pointer based code that could be optimized based on the theory introduced in this chapter. A compiler could allow for loop permutations on pointer linked structures (i.e. iterating row by row instead of column by column) by identifying orthogonal sets of confined components, and possibly also the application of complex loop analyses such as the polytope model that can assist with skewing and parallelization [39] of loops.

Section 3.1 gives definitions used later on in the chapter. Though some of the definitions are not new, they are given in order to make the chapter self contained. Confined components are introduced and the MINIMUM CONFINED COMPONENTS problem is proved to be NP-complete in Section 3.2. In Section 3.3 *CCDA*, a near optimal polynomial time algorithm capable of finding

confined components is introduced and discussed. Section 3.4 introduces orthogonal edge labeling, a concept that builds on confined components. Finally, theoretical and practical applications of the confined components are discussed in Sections 3.5 and 3.6.

3.1 Definitions

This chapter introduces new graph theoretic concepts, some of which are tightly related to grids. In this section we introduce the fundamentals that are needed to step by step build up the notion of confined components and the formal definition of grids. Note that for the remainder of this chapter, we are dealing with directed graphs unless otherwise specified. A component $G_C(V_C, A_C)$ as used here, is a subset of an existing digraph $G(V, A)$ s.t. $V_C \subseteq V \wedge A_C \subseteq A$. Note that the symbol $\veebar$ may be used as the symbol for *exclusive or*.

Definition 1. Unilaterally Connected Component (UCC)

A unilaterally connected component [7] is a subset of nodes and arcs such that for every pair of nodes p, q there is a path from p to q or from q to p (or both).

Definition 2. Exclusive Unilateral Component (EUC)

An exclusively unilateral component is a subset of nodes and arcs such that for every pair of nodes p, q there is a path from p to q if and only if there is not a path from q to p.

Definition 3. Traceable Component

A traceable component is a subset of nodes, $V_C \subseteq V$ and arcs $A_C \subseteq A$ that has a Hamiltonian path, i.e., there is a permutation of V_C, say $v_1, v_2, \ldots, v_r$ with $(v_i, v_{i+1}) \in A_C$ for $1 \leq i < r$, where $r \leq |V|$.

For grids, there exists an assignment of arc colors and numerical vertex labels such that in for instance the two-dimensional case, each vertex has two labels L_1 and L_2 (representing *row* and *column* indices) and each arc is assigned one of two colors C_1 and C_2 so the color represents *row* or *column* directions. We can define this more exact as follows:

Definition 4. A digraph $G(V, A)$ with vertices V and arcs A is called a grid if there exist an arc coloring C_1 and C_2 and a vertex labeling L_1 and L_2, such that:

- There is a maximum of two incoming arcs per vertex.
- All incoming arcs for a vertex $v \in V$ have different colors.
- There is a maximum of two outgoing arcs per vertex.
- All outgoing arcs for a vertex $v \in V$ have different colors.

and that for all $(u, v) \in A$ we have:

$$\begin{cases} L_1(u) < L_1(v) \wedge L_2(u) = L_2(v) & \textbf{if } (u,v) \text{ has color } C_1 \\ L_1(u) = L_1(v) \wedge L_2(u) < L_2(v) & \textbf{if } (u,v) \text{ has color } C_2 \end{cases} \tag{3.1}$$

In essence, this means that the labeling L_1 and L_2 take on the role of row and column index.

When we refer to grids in this chapter we may use the terms complete, sparse and directed grid in order to avoid ambiguity in the interpretation from the reader. A complete grid is a grid where the labeling relation is $L_n(u) = L_n(v) - 1$ instead of $L_n(u) < L_n(v)$. A sparse directed grid is what has been defined in Definition 4, and an example can be seen in Figure 3.1. An undirected grid is a directed grid where the arcs have been replaced with edges.

In the literature, the notion of grid is also often used for undirected graphs where vertices have a two dimensional index in $\{1, \ldots, n\} \times \{1, \ldots, n\}$ for some n, and vertices are adjacent when in one dimension, they have the same index, and in the other dimension, their index differs exactly one. If we direct all edges in such a graph to the right or down, we obtain a complete grid.

It is easy to build a sparse grid from a complete grid, by removing vertices and replacing the arcs to and from that vertex with arcs that maintain the same row and column connectivity.

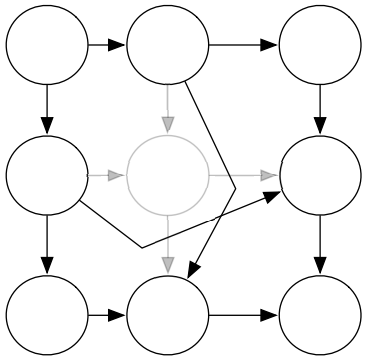


Figure 3.1: A Sparse Grid. As can be seen, the middle vertex of the complete grid is removed. A sparse grid is formed by replacing the removed element with arcs while maintaining the row and column connectivity of the surrounding vertices.

Definition 5. Directed Feedback Vertex Set

Given a digraph $G(V, A)$, where V is the set of vertices and A the set of arcs. A subset $W \subseteq V$ is called a feedback vertex set, if each cycle in G contains at least one vertex of W.

3.2 Confined Components

Confined components is an important new concept, that is related to directed grids. The exact relationship will be explored in Section 3.4. For now, we can summarize them as offering a graph theoretic approach that let us identify rows and columns in a grid structure, without resorting to vertex labeling or arc coloring.

When we refer to contraction, we specifically refer to edge contraction, the mechanisms of which are described in detail in for example [57]. Edge contraction is commutative, so we can say that contraction of a whole component is the application of the edge contraction operation on every arc within the component.

Terminology wise, after a set of components have been contracted, the resulting graph is in line with strongly connected component theory called a *condensate*, where each vertex represents a component in the original digraph.

Definition 6. Confined Unilateral Component

A confined unilateral component is a traceable component C of a *directed acyclic graph* (DAG) D, so that if the component is contracted, there are no induced cycles in the digraph D' resulting from the contraction of C.

Proposition 7. *A Confined Unilateral Component C is Exclusively Unilateral*

Proof. A confined unilateral component C has no cycles since it is part of a DAG. As C is traceable, by definition it contains an Hamiltonian path. Thus for every pair of vertices $(u, v) \in C$, $u \neq v$ in the component there is a path from u to v or from v to u, since there is a Hamiltonian path, but not both since there are no cycles. Hence, the confined component is an exclusively unilateral component as given in Definition 2. □

Lemma 8. *A decomposition of a DAG into a set of confined unilateral components will have a topological order*

Proof. Since the decomposition consists of confined unilateral components, there are no cycles in the condensate of these components. Since there are no cycles, the condensed graph is a DAG and all DAGs have topological orderings[1]. □

Let the labeling from Definition 4 correspond columns and rows, we call the label for columns L_C and the labeling for rows L_R. Let $L_C^{\max}$ be the maximum value of the labels for all $L_C(v)$ where $v \in V$ and $L_R^{\max}$ the maximum value of the labels for all $L_R(v)$, and $L_C^{\min}$ and $L_R^{\max}$ the corresponding minimum labels.

The vertices whose label $L_C(v) = L_C^{\max}$ are in the right most column, $L_C(v) = L_C^{\min}$ in the left most column, $L_R(v) = L_R^{\min}$ in the top row and $L_R(v) = L_R^{\max}$ in the bottom row.

A vertex v is considered to be above u if $L_R(v) < L_R(u)$ and below u if $L_R(v) > L_R(u)$. A vertex v is considered to be left of u if $L_C(v) < L_C(u)$ and right of u if $L_C(v) > L_C(u)$.

Proposition 9. *For a finite 2D sparse or complete directed grid $G(V, A)$, the set of components formed by contracting either all the rows or all the columns form confined components.*

[1]The fact that DAGs have topological orderings is well known and is discussed in many text books on graph theory and algorithms such as for example [21], which contains a proof for this on page 362.

Proof. Since the graph is finite, there must be a left and right-most column. The right-most column must form a confined component as the column has no vertices on the right hand side, and no further vertices above the top most vertex, or below the bottom vertex (in the same column), thus it only has incoming arcs, and therefore it is not part of any cycle. If the column is removed, a new column is made the right-most column, the previously removed column cannot form a cycle with the current one (since no arcs are going back to the rest of the grid), and must be traceable or it would not be a column in a grid. Since none of the columns then form cycles with other columns and all columns are traceable, the columns meet the definition of confined components. The same reasoning holds when replacing columns with rows. □

The following theorem was proven by *Hans L. Bodlaender*:

Theorem 10. *Decomposing a DAG into the minimum number of confined components is NP-complete.*

Proof. The problem clearly belongs to NP. To show it is NP-hard, we use a transformation from Directed Feedback Vertex Set. Let an instance of Directed Feedback Vertex Set be given, i.e., a directed graph $G = (V, E)$ and an integer L. Build a directed acyclic graph $H = (W, A)$ as follows. For each vertex $v \in V$, we take two vertices x_v and y_v, and an arc (x_v, y_v). For each arc $a = (v, w) \in E$, we take three vertices, $z_{a,1}$, $z_{a,2}$ and $z_{a,3}$. We add arcs: $(z_{a,1}, z_{a,2})$, $(z_{a,2}, z_{a,3})$, $(x_v, z_{a,2})$, $(z_{a,2}, y_w)$. Set $K = |V| + |E| + L$.

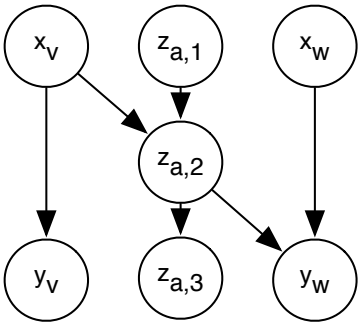


Figure 3.2: Construction: subgraph for arc $a = (v, w) \in E$

Note that H is acyclic: taking first all vertices of the form x_v, then all vertices of the form $z_{a,1}$, then all vertices of the form $z_{a,2}$, then all of the form $z_{a,3}$, and finally all vertices of the form y_v gives a topological order.

Claim. G has a feedback vertex set of at most L vertices if and only if H has a confined decomposition with at most K vertex sets.

Proof. $\Rightarrow$: Suppose G has a feedback vertex set $X \subseteq V$ with $|X| \leq L$.

Now, for each arc $a \in E$, we take the vertex set $\{z_{a,1}, z_{a,2}, z_{a,3}\}$. We say that this set represents a. For each vertex $v \in X$, we take two vertex sets, each with one element: $\{x_v\}$, and $\{y_v\}$. For each vertex $v \in V - X$, we take one vertex set with elements $\{x_v, y_v\}$. We say that this set represents v.

Clearly, each of these at most $|E| + |V| + L$ vertex sets induce a subgraph of G that has a Hamiltonian path.

We need to show that the graph obtained by contracting each vertex set to a single vertex is acyclic. First, note that for all $v \in X$, the vertex set $\{x_v\}$ has no incoming arc, so can never be on a cycle. Similarly, for $v \in X$, the set $\{y_v\}$ has no outgoing arc, so is not on a cycle. Now only consider the other vertices in the graph obtained by contracting vertex sets. A vertex that is a contracted set representing a vertex has only arcs to and from vertices that is a contracted set representing an arc, and moreover, if we have an arc from the set representing $v \in V - X$ to the set representing $a \in E$, then v is the head of a. Similarly, if we have an arc from the set representing $a \in E$ to the set representing $v \in V - X$, then v is the tail of that arc. Thus, if we have a cycle in the graph obtained by contraction, then this directly corresponds to a cycle in $G[V - X]$ which contradicts the fact that X is a feedback vertex set.

$\Leftarrow$: Suppose H has a confined decomposition with at most K vertex sets.

We say a confined decomposition is *fine,* if for each arc $a \in E$, we have a set $\{z_{a,1}, z_{a,2}, z_{a,3}\}$. We claim that there also is a fine confined decomposition with at most K vertex sets. We can obtain such a fine confined decomposition, by repeating the following steps:

- Consider a vertex set V_i that contains $z_{a,2}$ for some $a \in E$ but that does not contain $z_{a,1}$. Then, note that $\{z_{a,1}\}$ forms a one-element vertex set in the decomposition, as the vertex $z_{a,1}$ only has an arc to $z_{a,2}$ and no incoming arcs. If V_i contains an element of the form x_v, then we replace the sets V_i and $\{z_{a,1}\}$ by the sets $\{x_v\}$ and $V_i - \{y_v\} \cup \{z_{a,1}\}$. Otherwise we replace the sets V_i and $\{z_{a,1}\}$ by the set $V_i \cup \{z_{a,1}\}$.

- Consider a vertex set V_i that contains $z_{a,2}$ for some $a \in E$ but that does not contain $z_{a,3}$. Then, note that $\{z_{a,3}\}$ forms a one-element vertex set

in the decomposition, as the vertex $z_{a,3}$ only has an arc from $z_{a,2}$ and no outgoing arcs. If V_i contains an element of the form y_v, then we replace the sets V_i and $\{z_{a,3}\}$ by the sets $\{y_v\}$ and $V_i - \{y_v\} \cup \{z_{a,3}\}$. Otherwise we replace the sets V_i and $\{z_{a,3}\}$ by the set $V_i \cup \{z_{a,3}\}$.

We repeat the steps above while possible, and will obtain a fine confined decomposition with the same number or fewer vertex sets.

For a fine confined decomposition, vertex sets that contain vertices of the form x_v or y_v can be of the following types: there is a vertex $v \in V$ with the set of the form $\{x_v\}$, the set of the form $\{y_v\}$, or the set of the form $\{x_v, y_v\}$. Let $X \subseteq V$ be the vertices in V with a set of the form $\{x_v\}$ in the fine confined decomposition. Clearly, for $v \in X$, also the set $\{y_v\}$ is in the decomposition. So the number of sets in the fine confined decomposition equals $2\,|X| + (|V| - |X|) + |A|$, and hence $|X| \leq L$.

We will now show that X is a feedback vertex set in G. Suppose not. Then we have a cycle in $G\,[V - X]$, say with successive vertices $v_0, v_1, v_2 \ldots v_{r-1}, v_r = v0$. Now, for each i, $0 \leq i < r$, the vertex set $\{x_{v_i}, y_{v_i}\}$ has an arc in the set $\{x_{v_{i+1}}, y_{v_{i+1}}\}$. And thus, these vertex sets form a cycle in the contracted graph, which is a contradiction.

So, X is a feedback vertex set in G of size at most L. □

As H can be constructed in polynomial time from G, and as Directed Feedback Vertex Set [29] is NP-complete, the theorem follows. □

Apart from its application context, Theorem 3.2 is also interesting if we compare it with some classic results in algorithmic graph theory. Compare the notion of unilaterally connected components with strongly components: a strongly connected component in a digraph is a set of nodes such that for every pair p, q, there is a path from p to q and from q to p. Partitioning a digraph in its strongly connected component can be done in linear time with a well known application of depth first search (see e.g., [11, Chapter 22]. Many problems on directed acyclic graphs are efficiently solvable (e.g., [11, Chapter 24.2]. Determining if a digraph has a Hamiltonian path is known to be NP-complete [29], but is trivially solvable in linear time for directed acyclic graphs.

Although the Minimum Confined Components problem is NP-complete, it is relatively easy to decompose a DAG into confined components, where the number of confined components is not necessarily minimal.

3.3 Confined Components Decomposition Algorithm

We have devised an efficient greedy polynomial time algorithm that can decompose a DAG into confined components. The method devised is illustrated as Algorithm 1. This algorithm is optimal in terms of finding the minimum number of components, for trees, complete square and triangular grids; and works as expected on cyclic digraphs, i.e. it does not explode in complexity or incorrectly place nodes that are in a cycle in a component.

The algorithm starts at any source node and finds the heaviest path without joins (nodes with in-degree > 1). As metric for weight, we use the sum of the out-degree of all the branching nodes (nodes with out-degree > 1) in the component. Though one could use the number of nodes, the argument for using the sum of the branch node out-degree is that by maximizing the number of out-edges from a selected component, the in-degree sum in the rest of the graph is minimized when removing the component, thereby reducing the number of join nodes in the rest of the graph. Although not empirically evaluated here, this approach also turned out to work better in practice for the example graphs that the algorithm was tested on.

Only branches and sequences of nodes are placed in the path. Since no merge nodes are in these paths, the search space will be restricted to a tree. The heaviest path can be found with a DFS based heaviest path search, if a heavier path is found, the current heaviest path will be updated (when the search has reached a leaf). Though, the copy of the current path to the heaviest path does not look linear in the first case, it can easily be made so by implementing an incremental copy scheme. This could work by first allocating the extra space, and then copying backwards, stopping with the copy when the two paths join together. Thus, even though there may be a copy in every leaf, this copy will only copy nodes that have not yet been copied. So, the copy complexity in total is only $O(|V|)$.

When the heaviest path has been extracted, the nodes in that path are removed from the graph and any new source nodes (due to the node removal) are added to the list of sources and the computed heaviest path without merges is added to the list of confined components.

The algorithm also works in the reverse direction, by starting in sink nodes and constructing the components but switching merge nodes with branch nodes.

Note that the algorithm runs in polynomial time, this is easy to see as the algorithm perform DFS traversals and removes at least one node from the

Algorithm 1 Confined Component Detection Algorithm (CCDA)

```
1  def heaviestNonMergingPath(v, currentPath, heaviestPath):
2    # Stop at joins so we are bounded to a tree
3    if len(v.sources) > 1:
4      return
5    currentPath.push(v) # Add node to current path
6    # Increment the length metric for branches
7    if v.targets > 1:
8      currentPath.passedBranches ++
9    # Visit all targeted nodes
10   for tgt in v.targets:
11     heaviestNonMergingPath(tgt, currentPath, heaviestPath)
12   # Check if the path is longer than the known
13   # longest path
14   if currentPath.passedBranches
15     > heaviestPath.passedBranches
16   or heaviestPath.isEmpty():
17     heaviestPath = currentPath
18   if v.targets > 1:
19     currentPath.passedBranches --
20   currentPath.pop()
21
22 def FindConfinedComponents(G):
23   cc = []
24   sources = G.findAllSourceNodes()
25   for src in sources:
26     path = []
27     heaviestNonMergingPath(src, [], path)
28     cc += [path]
29     G.removeNodes(path)
30     sources += G.findNewSources()
31   return cc
```

graph for each DFS.

Theorem 11. *CCDA is correct and results in a set of confined unilateral components.*

Proof. There are two properties to be proven. Firstly, that each component will form traceable paths. Secondly, that the contracted graph will be cycle free. Let $G(V, A)$ be a DAG. Let $In(v)$ and $Out(v)$ be the in- and out-degrees of the vertex v. Let C_k be a component in the graph G and let us call the contracted graph $G_C = (V_C, A_C)$, where V_C is the contracted components and A_C is the arcs between the components.

Claim. Each component when selected is traceable.

Proof. First, let us rewrite the `heaviestNonMergingPath` function more formally as follows:

$$f(v,p) = \begin{cases} p & \textbf{if } In(v) > 1 \\ p \oplus v & \textbf{if } In(v) \leq 1 \wedge Out(v) = 0 \\ L(f(w, p \oplus v) | \forall (v,w) \in A & \textbf{if } In(v) \leq 1 \wedge Out(v) > 0 \end{cases} \tag{3.2}$$

where $v \in V$ is a vertex in G, p is a sequence of vertices from G, $\oplus$ is concatenation, and L selects the heaviest sequence of the list of sequence parameters.

The property to prove is that the result of $f(v, p)$ yields a traceable path (sequence of vertices).

The initial invocation of `FindConfinedComponents` invokes $f(v, \emptyset)$, with $In(v) = 0$. Then we have two cases: $Out(v) = 0$ or $Out(v) > 0$. If $Out(v) = 0$, the result will be $\emptyset \oplus v = v$, a single vertex and therefor traceable. If $Out(v) > 0$, all subsequent invocations will be $f(w, \emptyset \oplus v)$ or $f(w, v)$, or in other words have as argument a traceable path, together with vertex w, who is directly connected to the last node of that path.

Now assume f is invoked by f, with as argument a traceable path p and a node v, for which an edge exists in A connecting the last node in p with v. Then in case $In(v) > 1$, the result is p and therefore traceable. In case $In(v) \leq 1 \wedge Out(v) = 0$, the result is $p \oplus v$, and therefore traceable. In case $In(v) \leq 1 \wedge Out(v) > 0$, all invocations of $f(w, p \oplus v)$ have as argument a traceable path. The proof follows by induction. □

Claim. There will be no cycles in the contracted graph.

Proof. The second property follows directly from the fact that as seen above, any invokation of f is either with a vertex v whose indegree is 0, or with a path p whose last node has an edge to v. In case there would be a cycle, there would be an arc from the rest of the graph to a node v in the contracted component. This can only be the case if there would be an invokation of $f(p, v)$, with $In(v) \geq 2$. But in this case, v would not have been added to the contracted component. □

Since both the claims are valid the theorem holds. □

Theorem 12. *CCDA finishes in polynomial time.*

Proof. CCDA works by doing a DFS bounded by the tree formed by the cut-set of all $In(v) > 1$, this DFS search is $O(|V| + |E|)$. Whenever a DFS is complete, at least the source node is removed from the graph and placed in a component. Thus CCDA has a complexity of $O\left(|V|^2 + |V||E|\right)$. □

Theorem 13. *For trees, CCDA finds the minimum number of confined components and is therefore optimal.*

Proof. This is easy to prove, we first show that a decomposition of a tree into confined components must have at least as many components as there are leafs, and then that the algorithm will generate one component per leaf.

Claim. Decomposition into confined components will have at least as many components as there are leafs.

Proof. Assume that this would not be the case, then there would be at least one confined component where there were two leafs. However, such a decomposition is clearly not traceable and contradicts the definition of a confined component. □

Claim. CCDA generates exactly one component per leaf.

Proof. Since we are dealing with a tree, the algorithm will start with the root of the tree. The algorithm then performs a DFS, locating a traceable sequence that ends in a node that has out-degree 0 or in-degree > 1, however, since the graph in question is a tree, there are no vertices with in-degree > 1. Hence, the last node in the extracted component will be a leaf, but in order to reach that vertex, the algorithm will only descend through vertices that are not leafs. Hence, when starting the find longest non merging path algorithm from the root of a tree, the extracted component will contain exactly one leaf vertex.

When extracting one component, it is removed from the tree; this will not introduce any new leafs in the new graph, as a component will be removed starting with the tree's root, and not at any internal vertex.

The above reasoning applies to the further application of the subtrees formed by the component removal. □

Since a covering set of confined components must contain at least as many confined components as there are leafs in the tree, and the algorithm will generate exactly one confined component per leaf, the theorem follows. □

Theorem 14. *For complete (directed) grid-graphs, CCDA finds the minimum number of confined components and is therefore optimal.*

Proof. Consider an $m \times n$ grid where all the downward pointing arcs are labeled "*down*" and rightward pointing arcs are labeled "*right*", if there is no such labeling of the grid, then rotate it so that this labeling is valid.

Claim. Any trace in a grid must be a combination of down and right steps

Proof. Follows by definition of $m \times n$ grids. □

Claim. A confined component trace can consist out of arcs, either labeled right or down, but not a combination of these two.

Proof. Assume that this is not the case, then we have a (partial) trace p, q, r, where (p, q) is labeled *down* and (q, r) labeled *right*. In this case, there must be a vertex s, which is connected as follows: (p, s) labeled right, (s, r) labeled down. In this case, there is a path from p, q, r to s, and a path from s to p, q, r. These two paths however form a cycle between the two components contradicting the definition of a confined component. □

Claim. A confined component will have at most $\max(m, n)$ vertices in it.

Proof. This is trivial as a confined component will not contain arcs of different labels, the only components that can be constructed form parts of a row or a column in the grid. □

Claim. The algorithm will result in a set of components, all of length $\max(m, n)$.

Proof. Since the graph is a grid, the algorithm starts in the upper left corner of the grid (the only vertex whose in-degree is 0). The algorithm will then locate the longest non-merging path, this must either be the upper row or the left-most column. Assume without loss of generality that $m < n$, then the algorithm will find a row since each row is longer than each column. When this row of size n is removed, the grid shrinks to the size $\tilde{m} \times n$, with $\tilde{m} < m < n$. Further iterations will remove additional rows of length n and this happen while $\tilde{m} > 0$, until there are no vertices left to remove. □

Since CCDA will remove $\min(m, n)$ components of length $\max(m, n)$, the theorem follows. □

Theorem 15. *For complete triangular grids*[2]*, CCDA finds the minimum number of confined components and is therefore optimal.*

Proof. Follows in an analogous way to the proof of Theorem 14. What needs to be shown additionally, is that for the complete triangular grid there will never be a confined component including a diagonal arc. Assume this is not the case and the grid block has vertices p, q, r and s (named in clockwise order), and arcs (p, q); (p, r); (p, s); (q, r) and (s, r), where the diagonal arc is (p, r). From the component built by including (p, r), there are now paths through both q and s forming cycles with the component including (p, r), in turn contradicting the definition of a confined component.

Note that q and s cannot be part of the confined component at the same time (otherwise it is not traceable), and neither p, q, r nor p, s, r can form components (as shown in Theorem 14). Therefore, in the contraction of the graph, there will be at least two cycles because q and s will be contained in two different components. This contradicts Definition 6 and the theorem follows. □

Conjecture 16. *Finding the rows or columns in a given sparse grid is NP-complete*

Motivation: In some cases the set of rows or the set of columns in a sparse grid will be the minimum set of confined components. Therefore, finding these rows or columns is similar to finding the minimum number of confined components and hence most likely to be NP-complete.
As it is difficult to prove statically on a program that a data structure will not have any cycles, it is of interest to explore the behavior of the algorithm even on cyclic digraphs. We show here that for cyclic graphs, the algorithm will terminate in polynomial time and that any node that is in a cycle will not be placed in a component. The latter can be used to reject a graph for being cyclic during the runtime of a program.

Theorem 17. *Even for cyclic digraphs, CCDA is polynomial.*

Proof. As is the case for DAGs, at every outer loop pass, the algorithm will take one source node (in-degree = 0) and remove at least this node. The algorithm performs a DFS which is $O(|V| + |E|)$ in time, this DFS will terminate even in the presence of a cycle because a vertex in a cycle will be seen as a merging point in the current path. If there are no source node left in the graph (i.e.

[2]Triangular grids are defined in a similar way to complete square grids where each square is being tessellated into two triangles and the diagonal arc is pointing down.

the remaining nodes are parts of cycles), findMoreSources will return an empty list and the loop will thus terminate. Consequently, the complexity on cyclic DAGs is at most $O(|V|^2 + |V||E|)$. □

Theorem 18. *For cyclic digraphs, the CCDA will not embed nodes that are part of cycles in any component.*

Proof. We prove this by induction.

Claim. The first removed chain will not consist of nodes that are part of a cycle.

Proof. The algorithm will start at some source node (in-degree = 0), and remove a component without incoming arcs. A chain of nodes without incoming arcs cannot be part of a cycle. If this would be the case, then there would be an incoming arc to one of the nodes making that node's in-degree > 1. This contradicts the non-merging component property. □

Claim. A chain removed, assuming no nodes within a cycle have been removed before, will never be part of a cycle.

Proof. Consider a cycle, all the nodes on a cycles will initially have an in-degree > 0. The outer loop of the algorithm starts at a node whose in-degree is 0, and finds a chain of nodes with in-degree = 1. This chain of nodes cannot be part of a cycle. If this would be the case, one of the nodes would have an in-degree of at least 2, but this contradicts the non-merging component property. □

Since neither the first nor any subsequently removed components have nodes that are parts in a cycle, the theorem follows. □

3.4 Orthogonality

In this section we describe how confined components are used to define the notion of orthogonality for graphs. In fact this notion will allow us to infer grid-structures on any graph. Thereupon, this property can be used to transform a random graph traversal into a n-dimensional "grid" traversal. By doing so, these graph traversals can be transformed to n-dimensional loop structures (see Section 3.5). Orthogonality is based on labeling the arcs in the graph. All the arcs in the graph with identical labels are in turn grouped into the same arc set.

Definition 19. Orthogonal Graph
Let $G(V, E_1, E_2)$ be a digraph with vertex set V and arc sets E_1 and E_2. G is orthogonal iff $\forall p, q \in V : p \rightsquigarrow_{E_1} q \Rightarrow \neg(p \rightsquigarrow_{E_2} q)$. [3]

In general, this simply means that E_1 and E_2 are independent arc sets; it does for example not exclude branches or joins within the same arc set. Note also that if $G(V, E_1, E_2)$ is orthogonal, then by contraposition $G(V, E_2, E_1)$ is also orthogonal.

For discussing the results of the remainder of this section, we use the following notions. Let C_1, C_2 be two edge-disjoint vertex-covering confined component sets. Let E_1 be the set of edges in C_1 and E_2 the set of edges in C_2. Number all nodes in V according to their component order in C_2. Note that such an ordering exists as the contracted graph of the C_2 confined components is a DAG. Let $n(x)$ yield the number assigned to the vertex x. Let $f(x) = y$ such that $x, y \in E_2$. Let $out(x)$ be the out-degree of x with respect to E_2, that is for the vertex x, the number of out arcs in E_2.

Proposition 20. *Two edge-disjoint vertex-covering confined component sets C_1 and C_2 are orthogonal.*

Proof. Proof by contradiction: Assume the proposition does not hold, then we have $p \rightsquigarrow_{E_1} q$ and $p \rightsquigarrow_{E_2} q$. Contracting the set C_1 results in a singleton cycle formed by the edges of E_2, which contradicts the confined component assumption given in Definition 6. □

Lemma 21. *Two edge-disjoint vertex-covering confined component sets are ordered, that is:*

$$\forall\,(x, y) \in E_1 : n(x) < n(y) \tag{3.3}$$

$$\forall x, y \in \{v \in V : out(v) > 0\} : \begin{cases} n(x) < n(y) \Rightarrow n(f(x)) < n(f(y)) \\ n(x) = n(y) \Rightarrow n(f(x)) = n(f(y)) \end{cases} \tag{3.4}$$

Proof. Assume not, then either:
$\exists\,(x, y) \in E_1 : n(x) > n(y)$. Then because of edge disjointness of C_1 and C_2, (x, y) will be an edge between two different components in C_2. This clearly violates the ordering from the contracted C_2.

[3] With the notation of $p \rightsquigarrow_E q$ we mean that there exists a path from p to q in arc set E.

$\exists\,(x,y) \in E_1 : n(x) = n(y)$. Indicating that despite having a direct path in E_1 from x to y, there is a confined component in C_2 that includes this edge, contradicting the edge disjointness of E_1 and E_2.

$\exists x, y \in V : n(x) < n(y) \Rightarrow n(f(x)) \geq n(f(y))$. Then, clearly x and y are in different components of C_2, and either there is a cycle in the contracted graph of C_2 which is in contradiction with the definition of confined components, or $n(f(x)) = n(f(y))$. In the latter case there are edges in E_2 that are not part of any of the components in C_2, contradicting the definition of E_2.

$\exists x, y \in V : n(x) = n(y) \Rightarrow n(f(x)) \neq n(f(y))$. Then, there are edges in E_2 that are not part of any of the components in C_2, contradicting the definition of E_2. □

Definition 22. Strictly Ordered Orthogonality

We call two edge-disjoint vertex-covering confined component sets *strictly ordered orthogonal*.

Theorem 23. *The set of rows and the set of columns in a (sparse) directed grid are strictly ordered orthogonal.*

Proof. From Proposition 9 we know that the set of rows and the set of columns are confined component sets. Clearly these are also vertex covering.

Assume they are not edge disjoint, i.e. a an arc u, v is both in a row and a column. In that case there will be two colors assigned to the same arc which contradicts Equation 3.1. □

Theorem 24. *Let C_1 and C_1 be strictly ordered orthogonal, edge covering and let all the components have an Eulerian trail that is the same as the trace, then C_1 and C_2, may be represented as a (sparse) grid where the component sets C_1 is the set of rows and C_2 the set of columns (or columns and rows respectively).*

Proof. Consider Equations 3.3 and 3.4 from Lemma 21, and apply it on both sets. Let $m(x)$ be the number assigned to each vertex from its order in the contracted components of C_1. Then:

$$\forall (x,y) \in E_1 : n(x) < n(y) \wedge m(x) = m(y) \tag{3.5}$$

$$\forall (x,y) \in E_2 : m(x) < m(y) \wedge n(x) = n(y) \tag{3.6}$$

Note that, $<$ follows from the theorem and $=$ from the premise, as vertices involving the arcs of E_2 will be in the same component of C_2.

Assign a color C_1 to the arcs in E_1, and the color C_2 to the arcs in E_2. Then we have one color per arc, or $\forall(u,v) \in E : C_1(u,v) \veebar C_2(u,v)$. We can then rewrite Equations 3.5 and 3.6 as:

$$\begin{cases} n(u) < n(v) \wedge m(u) = m(v) & \textbf{if } C_1(u,v) \\ m(u) < m(v) \wedge n(u) = n(v) & \textbf{if } C_2(u,v) \end{cases} \tag{3.7}$$

which is equivalent to Equation 3.1.

Because the components have intra-component Eulerian trails that is identical to the intra-component traces, there can be at most one intra-component outgoing arc per vertex and at most one intra-component incoming arc per vertex. Each vertex can have at most one incoming and one outgoing arc of color C_1 and at most one incoming and one outgoing arc of color C_2. □

We provide a simple algorithm capable of verifying whether a decomposition is strictly ordered orthogonal. If such an algorithm is going to successfully run, it needs to ensure the following attributes.

- The graph must be shown to be initially cycle-free.
- For each component set, the components must be confined.
- For each component set, the components must be edge disjoint from the components in the other component sets.
- For each component set, the components must be vertex covering.

Algorithm 2 details the straight forward way to accomplish the verification.

Algorithm 2 Strictly Ordered Ortho. Verification Algorithm (SOOVA)

```
1  def HasCycles(G):
2    SCCs = Tarjan(G):
3    if length(SCCs) == 0:
4      return False
5    return True
6
7  def Contract(G, cs):
8    G2 = Graph()
9    Map = {}
10   for comp in cs.components:
11     v = Vertex()
12     G2.vertices += [v]
13     for v2 in comp.vertices:
14       Map[v2] = comp
15   for e in G.edges:
16     if Map[e.src] != Map[e.dst]:
17       G2.edges += [Edge(Map[e.src], Map[e.dst])]
18   return G2
19 # Given graph G and set of decompositions CC
20 # True iff the decomp is strictly ordered orthogonal
21 def IsDecompStrictlyOrderedOrtho(G, CC):
22   if HasCycles(G):
23     return False
24   for cs in CC:
25     cg = Contract(cs)
26     if HasCycles(cg):
27       return False
28     for e in cs.edges: # Check if edge disjoint
29       if e.tag == None:
30         e.tag = cs
31       else:
32         return False
33     for v in G.vertices: # Check if vertex covering
34       if v not in cs.vertices:
35         return False
36   return True
```

For Algorithm 2, the following two theorems can be proved in a straight forward manner:

Theorem 25. *The SOOVA algorithm is correct for* simple digraphs[4]*.*

Proof. In order to prove this, we need to show that the algorithm identifies a graph to adhere to the strictly ordered orthogonality property.

Claim. The algorithm detects cycles in the full data structure.

Proof. At the very start of the algorithm, Tarjan's SCC algorithm [48] is executed on the whole graph. Tarjan's SCC algorithm finds strongly connected components. There exists strongly connected components in a graph iff there are cycles in the same graph. Hence, the claim is valid. □

Claim. The algorithm detects that each component set is confined.

Proof. By contracting each component set, and finding cycles in the contracted graph, the premises of Definition 6 holds. □

Claim. The algorithm finds shared edges between the components and proves edge disjointness.

Proof. For all the edges in a component set, the algorithm will check if the edge has a tag (it is assumed that no tags have been set when the algorithm starts). If no tag is set a tag will be set for the next outer iteration and component set check. Since all edges in previous component sets are tagged when the next component set is checked, any edge in multiple component sets will be detected. □

Claim. The algorithm detects that each component set is vertex covering.

Proof. By iterating over every vertex, and checking if it is part of the current component set, the claim holds. □

Since all the claims are valid, the theorem follows. □

Theorem 26. *Given L component sets, verifying that a digraph is strictly ordered orthogonal takes $O(L * (|V| + |E|))$ time.*

[4]It should be trivial to device code that iterates over all edges, finding and eliminating multi edges and loops

Proof. The main algorithm has two top level items, a cyclicity check known to be $O(|V| + |E|)$ and a loop over the L component sets. The loop contains the following steps: *contract*, *cyclicity check*, an *iteration over all edges* and an *iteration over all vertices*. Contraction in turn is $O(C + |E|)$, where C is the number of components. However, a component has at least one vertex, so $O(C) \leq O(|V|)$. Cyclicity check is obviously $O(C + E')$ where E' is the number of edges between the components, worst case there are no edges within the components, and $O(E') = O(|E|)$. The remainder is the iteration over all edges $O(|E|)$ and the iteration over all vertices $O(|V|)$.

Summing the complexities together yields the following complexity $O((|V|+|E|) + L * (C + |E| + C + E' + |E| + |V|)) = O(L * (|V| + |E|))$ □

The algorithm is mostly parallel, we discuss the issues with parallelizing the algorithm informally here. While, being inherently a depth first search problem, for cycle detection there are some parallelizable algorithms (for example [14]). For this algorithm, the cycle detection will be the main bottleneck as the remaining steps are relatively easy to parallelize.

Edge disjointness checks are completely parallel and each edge can be checked concurrently under the assumption that the visitation check and the tagging are atomic together. This can for example be accomplished using either locks, transactional memories or *test and set* instructions. Parallel behavior here is $O(|E|/P)$ in the best case, but as the test and check must be serialized and if every single edge is shared with all sets, the complexity will instead be $O(L * ((E')/P))$ where E' is the number of edges per set. However, if this test fails the algorithm can immediately report that there are overlapping edges, so in practice it should just signal the failure and return, so the complexity is still $O(|E|/P)$.

Contraction of the components is also essentially parallel, the initialization of each vertex in the contracted graph is obviously parallel. The addition of edges between the vertices may be parallel assuming proper atomics or locks are used (such atomics do of course serialize the access to a specific component vertex). In a naive approach with an evenly distributed edge count between the components, the parallelism is bounded by the number of components in the system and the execution time of the componentization should be in the order of $O((|V| + |E|)/C)$, where C is the number of components.

Assuming a worst case cycle detection time of $O(|V|log|V|)$ as given as a sequential time by [14], we get the total execution time as $O(|V|log|V| + |E|/P + (|V| + |E|)/C + ClogC)$.

Note that for all practical applications, L is a per data structure constant, and the running time will be linear with the size of the data structure in terms of

the number of vertices and arcs.

With the shown ordering of the confined components in a strictly ordered orthogonal graph, it is important to note that if a data structure is mapped into a grid based on the confined components, it is possible to maintain the data dependencies of the original traversal order.

3.5 Special Grids

For the theory introduced in this chapter there are both theoretical and practical applications. Already discussed is the notion of strictly ordered orthogonality, which is an application of the confined components introduced in this chapter. Strictly ordered orthogonality has been defined with the intention of pointer traversals being embeddable in n-dimensional arrays, where the number of edge disjoint vertex covering sets of confined components represents the dimensions in the array.

In terms of shapes, there are two grid types that we are interested in discussing: these are square grids and triangle grids. In both cases, these grids can be directly implemented with pointers, yielding dense mesh implementation. However, in many applications, computations are not defined on the direct implementation of these grids, but rather in an indirect way and on their adjacency matrix representations. These are especially common in high performance code libraries. For instance, in finite element applications the elements are typically loaded as a triangular or tetrahedral direct grid structure and after assembling the stiffness matrix, the computations switch to using an adjacency matrix. Note that in general, adjacency matrices are basically sparse rectangular grids.

3.5.1 Complete Rectangular and Triangular Grids

Strictly ordered orthogonality of complete square and triangular grids can easily be computed in polynomial time, as shown in Theorems 14 and 26.

An important property of square grids is that there is a two dimensional embedding of such grids so that pointer traversals can be translated into counted loops, with one index per dimension. However, for complete triangular grids (see Figure 3.3), the situation is a bit more complicated. This comes from the introduction of row alignment issues. For example, if the triangular grid is defined as given in the gray graph in Figure 3.3, one of the orthogonal arc sets is the set of diagonal arcs, pointing backwards in next row, this introduces back-dependencies that we want to avoid. This can either be avoided by con-

structing the grid in a way where this will not happen (i.e. as the white graph in the same figure), or by skewing the grid, so that the diagonals become the down arcs. Note that this is not a direct embedding and empty elements need to be inserted on the edges after the skewing to map this directly.

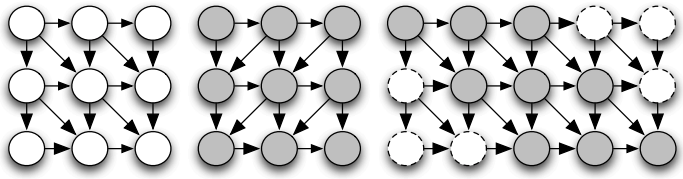


Figure 3.3: Triangular Grids. The fully white grid has the right and down directions orthogonal, the gray grid has its right and diagonal directions orthogonal. In the right hand graph, the middle graph has been skewed and empty elements inserted to ensure that the orthogonal directions line up and can be traversed by modifying only one iteration index.

Triangular grids have a three dimensional counterpart called a tetrahedral grid. Tetrahedral grids can also be defined in a way that allows for the determination of three orthogonal arc sets, for a simple example with twelve tetrahedrons, see Figure 3.4. Iterations over these arcs can then also be mapped into indexed loops where three indices are used. Essentially, a right angled tetrahedral grid can be mapped into a set of cubes (forming an overlaying cube grid), the dimensions of the cube represent the orthogonal arc sets which can be mapped to a single index increment and traversals in the remaining arcs can be replaced by a incrementing either two or three dimensions. Note that skewing needs to be handled in an even more intricate way than was the case for the triangular grids, and may not always be possible.

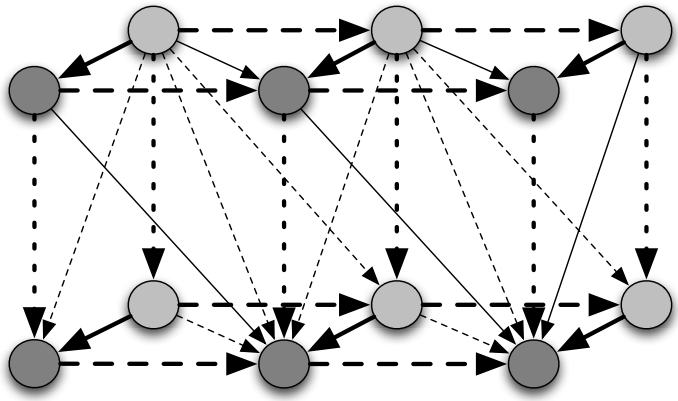


Figure 3.4: Orthogonal Tetrahedral Grid Building Element. The edge styles represent the different dimensions in the grid. In this case, the fat solid lines, the fat long striped lines and the fat short striped lines indicate the orthogonal dimensions in this grid (i.e. depth, horizontal and vertical dimensions).

3.5.2 Sparse Square Grids

Most sparse square grids arise from adjacency matrices in the code. These grids are strictly ordered orthogonal as defined in this chapter, and the underlying grid may be easily decomposable (in orthogonal arc sets). There are two options for decomposing these grids, one is to attempt decomposing the sparse grid itself, the second option is to map the sparse grid into the direct implementation and to decompose the direct grid[5].

Deriving the mapping to the direct grid is not straight forward and is rather artificial with respect to the algorithms working on the grids (row wise traversals become iterations on vertex neighbors, and column traversals become iterations over the vertices), i.e. a traversal over a sparse grid's rows and columns becomes a traversal over the direct grid's rows and columns and all neighbors of the visited nodes. Note that, in order to derive a mapping to the original complete grid we would need to find the rows and the columns in the adjacency matrix. This means that the decomposition of the direct grid in

[5]A direct implementation is a direct representation of a matrix in matrix-form; this should be contrasted to indirect solvers that map a matrix into an adjacency structure

strictly ordered orthogonal arc sets is dependent on the decomposition of the sparse adjacency matrix grid.

Presumably, for analysis purposes, we would want to insert empty fill elements in the sparse square grid in order to be able to embed it into a two dimensional grid. This is similar to how the triangular grid with back pointing dependencies was handled (bottom graph in Figure 3.3), except that the empty fill elements would also be inserted between the elements (not only outside the main grid). This insertion strategy serves as a way of applying optimizations designed for codes working on direct grids to codes working on indirect grids.

3.5.3 Exploiting Knowledge on Pointer Types

In many cases recursive pointer types are used to implement graph structures. This is important since the programs often constrain the purpose of each pointer field to point in a specific dimension. We can use this property to infer a decomposition into potentially orthogonal arc sets. The orthogonality assumption can in turn be verified with an efficient algorithm (see Theorem 26). For example, in a sparse matrix-vector multiply code, a common implementation is to use a record containing two pointer fields (right and down). This record may in turn be used to construct a graph as the one illustrated in Figure 3.5. Note that without any kind of verification of the assumed shape, it would not be correct to assume that the structure represents a matrix, since any pointer may alias the other pointers in the structure or may be part of some other structure (e.g. a tree or some generic DAG). In such cases, embedding in a two dimensional grid would not be a good choice.

We can also use the property of directed arc chains to eliminate pointer fields from the analysis and support anti-parallel feedback arc sets when such sets exist in a pointer structured graph. Despite the fact that such arcs introduce cycles (for example the set of all up and left pointers form an anti-parallel feedback arc set in a bidirectionally linked grid, that is, *left* is anti-parallel to *right*, and *up* is anti parallel to *down*, and removing these pointers will leave the remaining graph as a DAG), verifying that a named pointer field is anti-parallel with another pointer field is obviously linear with the number of vertices in the graph.

3.6 Potential Applications

Confined components and the orthogonality of arc-sets have several compiler applications that could almost be directly implemented in many existing com-

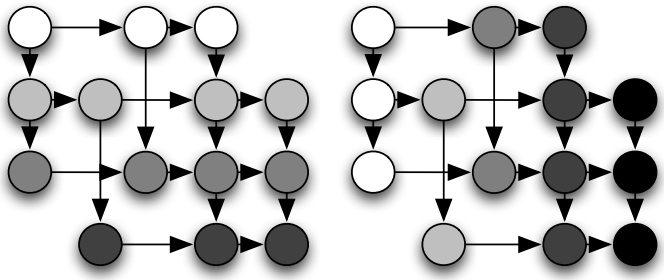


Figure 3.5: Strictly Ordered Orthogonal Decomposition of a Sparse Square Grid. Each confined component in the decomposition is indicated using a separate shade of gray.

piler tool chains. The fundamental area where the concepts can be applied is the optimization of pointer structured code and data structures. This type of optimization, where for example objects are being moved around or pointers are added to or removed from a structure, is called restructuring.

It should be noted that in general it is not possible to restructure data during compile-time since data in most cases is read from files that the user may modify. Because of this, restructuring of data needs to be done during run-time; but the compiler may help by analyzing the program and data structures in order to assist with the generation of restructuring code, or by restructuring the data structure types themselves (e.g. eliminating pointers in a structure or replacing them with indices).

There are three key problems that need to be solved in order to restructure pointer-based data structures. These are data structure memory *layout*, chained pointer *dependencies* and the pointer *aliasing* problems.

In the first case, when a pointer based structure is created (often involving a recursive data structure), the order in which elements are read from file will in many cases impact the memory order of the elements. An example that illustrates this is a common pattern from sparse matrices, where a structure is used for each element in the matrix. This structure is then usually linked to the next in row and the next in column elements. Depending on how the system's standard memory allocator works (e.g. malloc), the elements will be created one by one in an increasing memory order. If the order of the elements are the same as the logical structure, that is element $(0, 0)$ first, then $(0, 1)$

and finally (m, n), the matrix elements will be allocated in a row-wise order since the file the matrix is read from is ordered in that way. If the elements in the file are unordered the right element pointer will not necessarily point to the next physical element.

For the problem relating to chained pointer dependencies, it should be obvious that in order to access an arbitrary element within a pointer linked structure, a linear traversal of the pointer chain is needed. Such dependency chains directly impact parallelization and vectorization opportunities as a pointer chain cannot easily be split up through a divide and conquer like pattern.

The aliasing problem is dealing with the question of whether two pointers can point to the same object. Some languages such as Fortran make this implicit, others like C allow the programmer to have a more relaxed notion and to tell the compiler when there will be no aliasing. In many cases this is not practical, so several inter-procedural algorithms for doing points-to analysis have been devised such as Bjarne Stensgaards analysis [47] in order to automate the aliasing resolution. Such algorithms typically classify pointer dependencies as *will not alias*, *may alias* and *will alias*. This allows the compiler to eliminate redundant pointers and assume that objects are independent in many cases.

These three problems (order, dependencies and aliasing) are all very interesting by themselves, and various solutions have been developed that each deal with one of the problems and not the others. As will be shown here, the confined components and the notion of orthogonal edge sets may help to deal with all three of these problems.

For the applications discussed in this section, we predefine a number of types, though we only show the pointer fields in the types as these are the relevant fields for the graph structure.

```
typedef struct cell_t {
  cell_t *right;
  cell_t *down;
} cell_t;

typedef struct {
  size_t rows;
  size_t cols;
  cell_t **row;
  cell_t **col;
} matrix_t;
```

When the types are used in the applications below, we assume that it is known that the rows and the columns are vertex covering confined component

sets. Implicitly from this, we are aware that the recursive pointer chains will end in NULL at some point (or the pointer graph would not be a DAG).

We must also stress, that control flow information needs to be considered by the compiler before doing any transformations like the ones mentioned. For example, the component traversals may depend on additional variables and for example loop interchange may not be suitable without proper control flow information. Such control flow needs to be consider together with the orthogonality and confined component formalisms.

3.6.1 Linearization

For large pointer linked data structures, memory bandwidth may be a severely limiting factor. This is especially problematic when data structures visited in sequence are not adjacent in memory. The memory system will in many cases have to fetch unnecessary data outside the structure that is being loaded, that are not part of the next object. In addition to this, pointer based structures are not directly position independent (since they contain pointers). Both of the issues may be alleviated, by applying linearizing transformations (as for example described in [53]).

For a pointer linked data structure, a linearization pass would relocate the objects into an array, where the recursively typed pointers may be replaced with array indices. The transformation, while obviously being trivial to apply on a singly linked list, is non-trivial to apply on complicated pointer linked structures. For example, in the linked list case, we can simply traverse the list and relocate the objects, but for an object with more than one recursively typed pointer the object may represent any kind of graph structure, so the order of the linearization is not trivial, and it may not make sense in many graphs.

By showing a pointer linked structure to be orthogonal, we can apply the linearization step on two-dimensional grids. This is done by linearizing based on one of the dimensions (i.e. vertical or horizontal) in the grid. Figure 3.6. illustrate the transformations on the data structure.

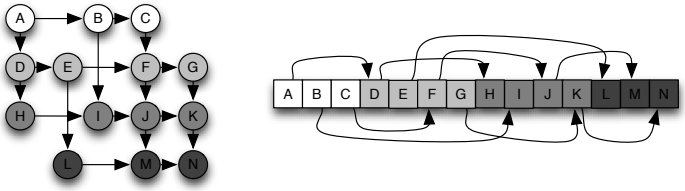


Figure 3.6: Linearization of an orthogonal grid. The left figure shows the data structure's logical layout (the nodes can be spread out on any kind of location), the right hand graph shows the structure in its post linearization physical layout.

Consider the following code iterating over the matrix type defined above:

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (p) {
    p = p->right;
  }
}
```

In order to exploit the linear behavior of the right pointer member, we make two transformations on the code, the first step moves the inner loop into a pointer increment based iteration:

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (WITHIN_ARRAY(p)) {
    p ++;
  }
}
```

The next step is to rewrite the while loop into a for-loop:

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  for (int i = 0; i < ARRAY_LEN(p) ; i ++) {
    // Replace dereferences of p with p[i]
  }
}
```

In order to accomplish this, the language or the optimizing runtime would need to provide information on how to determine array membership and lengths. Although, the C-language does not provide this information, the optimization relies on the fact that confined components have already been detected. Therefore assume that this information is available. As the pointer linked structures are typically built dynamically (by for example loading a file), it would be necessary to take this into account. We suggest that the compiler should be able to determine when a structure is complete, either through analysis or by programmer intervention.

3.6.2 Advanced Pointer Elimination

While the linearization mentioned earlier provided a simple type of pointer elimination, it still does not eliminate pointer fields in the structures. If pointers are unnecessary they will take up memory bandwidth, therefore it is of interest to get rid of some of the pointers completely.

A related problem is the position dependencies of the data structure, such dependency does for example prevent the offloading of computations within an heterogeneous system (such as for example GPUs) that work with different address spaces. Consequently, it would be interesting to eliminate unused pointer fields in a structure in order to save memory bandwidth and to make the structures position independent.

If pointers are used to traverse a chain, they can be indexified by building on the linearization step mentioned previously. Consider a pass through the data structure that replace all the pointers with indexes within the linearized structure instead. This structure is immediately position independent and it is now relatively easy to move the pointer based structure to a GPU.

Such transformations require access to the linearized structure's base pointer, and this may not always be possible. We could solve this by cloning functions, adding extra arguments with the base pointers (much like the work on automatic pool allocation, where pool pointers where automatically added to the function's parameter lists).

In order to replace the pointers within the linearized region, we need to introduce new types.

```
typedef unsigned long cell_idx_t;

typedef struct cell_t {
  // Right eliminated
  cell_idx_t down;
} cell_t;

typedef struct {
  size_t len;
  cell_t elements[];
} cell_array_t;

typedef struct {
  cell_array_t *cells;
  cell_idx_t start;
  cell_idx_t end;
} cell_ptr_t;

typedef struct {
  size_t rows;
  size_t cols;
  cell_ptr_t *row;
  cell_ptr_t *col;
} matrix_t;
```

The cell array can be used by the linearization pass to store all the cell objects. This is used by the cell pointer type, which can be used when dealing with cell pointers on the stack, or in other types, the cell pointer type has a start index and an end index in order to signal the start and end of a confined component within the array (in this case, the components are either rows or columns).

Assuming that the cell type is responsible for the majority of the data traffic, the setup shown here will reduce the memory bandwidth requirements by eliminating one pointer field (we can potentially also reduce the size of the index type to 32 bit if the application can be limited to work with 4 billion connected cell objects at most). In addition to reducing the bandwidth requirements, the matrix type will be mostly position independent (the only exception being the cell array pointers, but those can be reduced in numbers

by for example factoring out the array pointer from the cell pointer type and storing it separately in the matrix type).

Assuming that the application only stores a single double value in the cell objects the transformation has eliminated almost a third of the memory bus traffic stemming from the matrix traversals.

The suggestions in this section do depend on the linearization optimization described in the previous section. As such, it requires development of language constructs or control flow analysis that can determine when a structure can be rewritten in the given form (for example, the compiler must be able to determine that no additional objects are inserted in the structure and that the *right* pointers are constant). In order to execute a pointer based program on a GPU, additional technology would be needed that could generate working GPU code for parts of the program, while GPU compilers naturally exist and are well used. Some work has been carried out in the area of allowing unmodified C-code to execute on a GPU such as for example [2] that describes a compiler capable of generating CUDA code from dense C-based loops (that can be analyzed using the polyhedral model). We are not aware of any compilers that can take a sparse pointer based C-code and generate GPU code for parts of the program. As such, we believe that linearization in combination with advanced pointer elimination techniques could be a powerful transformation opening up the possibility for GPU based computation on some pointer based codes.

3.6.3 Replacement Algorithms, Garbage Collection and Leak Detection

Replacement algorithms and policies have been extensively studied by others, one of the most well known is the Least Recently Used (LRU) replacement policy that will replace the oldest reference (in the cache or Translation Look aside Buffers (TLB) for example) whenever the cache unit is full. Although the LRU policy is difficult to implement in hardware, many of the cache replacement algorithms are derived from LRU such as for example the pseudo LRU policy and others [13, 18, 42]. However, LRU and its derivatives are not optimal and it is easy to write code that will suffer if LRU is used. The optimal algorithm for replacement policy was described by Belady[3] and can be described as replacing the unit that will be needed the furthest time in the future, this however requires that the future is well known and can only be used as is in some severely restricted cases.

Related to the concept of cache replacement algorithms is the notion of garbage collection and leak detection. These concepts are similar to cache

replacement in the aspect that they are also dealing with reachability, although for replacement there is not necessarily a strict reachability requirement but rather a temporal requirement that the replaced objects should not be reached soon.

In either case, by applying the confined component detection and matching the links to a *successor* component within the iteration space in the code, we should be able to apply an informed replacement policy, or run a potentially efficient garbage collection / leak detection algorithm. In the former case, components that are exited by some iteration will be evicted from the caches, and in the second case, a component that is exited during an iteration could for example have a component-reference count reduced. Note that the common reference counting issues where cycles need to be specially handled does not apply in this case as components by definition are not part of cycles when contracted.

Consider the following line of code:

```
p = p->down;
```

In this case we assume that *down* always leave a component, though it may be possible to have more complex exit criteria we have at this moment not explored the area and as such this is left as future work.

Consider the act of leaving a component, it is likely that the previous component is no longer needed for some time, in this case the entire component can be evicted from cache. If the hypothesis that the down dereference leaves a confined component is correct, it may be suitable to transform the line into the following:

```
p_tmp = p;
p = p->down;
flush_component(componentof(p_tmp));
```

Naturally, whether or not this transformation makes sense would depend on the component, cache and TLB sizes. In this case, when a component is flushed, one stack root pointing into some compoenent is also replaced. If the code leaves a component, it may be necessary to consider what this entails for garbage collection and leak detection (the latter being similar but more of a debugging tool for gc-free systems).

In these cases it may be interesting to consider whether reachability information should be computed on component graphs instead of individual

objects. Essentially, this could potentially allow higher level GC and leak detection systems. The performance of these systems may be improved as they would no longer need to scan through every single heap object. Instead they can walk the contracted graph of components, which hopefully contains fewer vertices than the full graph.

The following example illustrates how some object pointer is declared on the program stack and then used in a reference counting system[6]. The difficult part is obviously not in the actual reference counting, but rather in the detection of the fact that p is pointing into a component of some sort.

```
{
  node_t *p = ...;
  inc_ref(componentof(p));
  ...
  dec_ref(componentof(p));
}
```

In order for this to work properly a runtime function for determining the component of an object would be needed. None of these applications rely on linearized components as described in section 3.6.1, although it should be noted that linearization would probably simplify the runtime.

3.6.4 For-each Detection and Loop Interchange

Given that the confined component sets may specify dimensions of a graph, it may be possible to determine that loops are actually for-each loops. Especially in the case of orthogonal arc sets, it may be possible to permute some loops that iterate over the main dimensions of the orthogonal structures. For example, if the rows are linearized and a loop is found to be traversing the columns, it would be better if the loops where interchanged and the iteration done over the rows instead. The exact safety of these transformation would naturally depend on the control flow of the application, and whether or not all row and column headers can be derived.

Consider the following loop, which is representative to some sparse matrix codes:

[6]Note, that we don't have to bother about the issues reference counting have with cycles since by definition, the confined components are cycle free.

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (p) {
    p = p->right;
  }
}
```

Consequently, if the compiler would hypothesize that row[] contains all roots in a vertex covering set of confined components formed by following the right pointers, and similarly that col[] contains all the roots forming a vertex covering set set of confined components with the down pointers. By determining that the two sets are orthogonal, it is possible to do loop interchange on the the loop. Then the loop can be transformed into:

```
for (int i = 0 ; i < m->col_count ; i ++) {
  cell_t *p = m->col[i];
  while (p) {
    p = p->down;
  }
}
```

This assumes that there is a way to bind the *col count* and *row count* variables to the actual array lengths. This is implicit in many programming languages (but not in C).

Like normal loop interchange, this optimization primarily targets the memory order of the dereferenced elements in the matrix. For sparse matrices, the memory order is typically dependent on the input to the program, so a good solution in this case would be to emit multi-versioned code (one for row major and one for column major orderings).

In addition to this, parallelism in the outer loop could potentially be exploited as it is clear that the inner loops are independent of each other. The code would then be trivially transformed into the following sequence:

```
parallel_for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (p) {
    p = p->right;
  }
}
```

3.6.5 Implementation Issues

As it is computationally expensive to detect confined components, we cannot expect that an implementation would try to do this during compile- or runtime without having constraints on how the components may be built.

We could assume that for example orthogonal edges will be different pointers in a structure (e.g. right and down pointers in a sparse matrix). Although subsequently verifying this during runtime could potentially be expensive as there is no conceptual difference between say a sparse matrix node and a binary linked list node. Both have two pointers embedded in themselves and cannot be distinguished from each other. Two of the most obvious ways to overcome this is to let the programmer specify additional meta-information through attributes or pragmas in the source code, or through the less general way where the compiler simply analyzes the language of the source code. If it finds fields named right and down it makes one assumption, but if it finds fields named right and left it makes another.

3.7 Summary

In this chapter we introduced a new graph theoretical concept, the confined component and showed several applications that could be used in a compiler (in combination with a run-time system). Though the MINIMUM CONFINED COMPONENTS problem was shown to be NP-complete, this restriction does not apply to practical cases where we do not need to find the components, but we have only to verify that we have a given decomposition of the components.

An efficient algorithm able to find the vertex covering sets of confined unilateral components was introduced. This algorithm was proven to be correct and optimal on some important graph shapes.

Strictly Ordered Orhogonality was introduced and an algorithm was given that is able to verify an orthogonality hypothesis in polynomial time.

We believe that a compiler that would take this theory into account would open up new venues into, for example, automatic parallelization of sparse irregular pointer based codes.

Chapter 4

Pax C

The shape of pointer linked data structures is the result of two key aspects: *control flow* and *input data*. For example, a recursively typed structure such as the one shown in Figure 4.1, with more than one pointer[1] in the structure may describe any kind of graph shape[2]. In the example, the type *Foo* may represent for example a binary tree, a sparse matrix node, a doubly linked list, a DAG or a generic graph. In fact, there is no way to infer what kind of shape the linked structure represents without taking the control-flow and, if necessary, the input data into account.

```
struct Foo {
  struct Foo *a;
  struct Foo *b;
};
```

Figure 4.1: Struct with two recursivelly typed pointers. It can represent any kind of graph configuration.

While there have been many attempts to alleviate this problem. Most systems have either relied on low or high level approaches. The low level approaches deal with methods such as data remapping and pointer aliasing ([12, 36, 38, 47, 54]), and high level approaches deal with the shape of the

[1] As C makes arrays and pointers identical, this is not entirely correct. But for now, let us consider pointers to be the same as references.

[2] By treating multiple linked structs as being the same vertex in a graph, we can construct arbitrarily complex data structure shapes as long as the structs have two or more pointers. Single pointers can only link together straight chains, cycles and lassos.

structures such as whether a data structure is a tree, a DAG, or a generic digraph ([17, 19, 24, 26, 41]). However, the low level optimizations are not powerful enough to make use of specific data structure characteristics, and the high level approaches are either too costly to implement or require too much user involvement. Also the high level approaches lack the ability to provide runtime support to automatically perform analysis at runtime or to carry out runtime verification whether the higher level properties are violated.

Another issue is the more subjective quality of ease of programming. While high level languages may add advanced grammars to describe different data structures, these grammars have in many cases been too far abstracted from the normal programming language.

In this chapter we introduce a C programming language extension named Pax C[3] which combines the efficiency of the low level approaches with the expressiveness of the high level approaches, while offering an easy model to program. At the same time, the additional code and attributes written by the programmer declares the *intent* of the pointer linked data structures to the compiler, and the conformance of the control flow to this *intent* can be verified at either compile or run time.

We have identified a number of properties that are important for the optimization of pointer linked structures. Firstly, *coverage* of linked data structures is a property that describes whether a specific path in a data structure visits all elements. This property enables *foreach optimizations* of pool allocated data structures. These optimizations serve two primary purposes, they ensure temporal locality and improved cache behavior, and they enable data parallelism within the pool.

Secondly, *disjointness* of linked data structures defines whether different paths are non overlapping or not. The property is closely related to aliasing, but we see it as a more high level property, where aliasing refers to individual pointers, and disjointness to the larger structure. For example, different branches in a tree are disjoint. Disjointness is useful for determining parallelism in the case where there are sequential dependencies.

Thirdly the *direction* of pointers determines whether two pointers are anti-parallel or if an object may be reachable through a combination of different pointers. The direction notion can improve aliasing analysis and enables the transformation of pointer chasing code into index increments (and for the anti-parallel pointers as decrements). This property is also usable for inferring disjointness in some cases. Unlike the coverage and disjointness properties,

[3]Pax C was in the early stages an abbreviation for *pointer axiom C*, though it should not be considered to actually mean this anymore.

it does not apply to traversal paths through the pointer structures but to pointers in the structure.

Fourthly we have the *firmness* property. With firmness we are dealing with how robustly a structure follows the other properties over time. For example, during construction, a pointer based structure may not respect that two pointers are supposed to be anti-parallel, but after the construction the property will be valid. Firmness is a temporal quality, and is thus expressed in the code, while the other properties are spatial properties that are associated with the data types in the program. Firmness can for example be expressed as either static or dynamic pointer structures, where in the first place, the pointers building up the backbone in the linked structure are constant, and in the dynamic case the pointers may change to point at other objects within a covering set of object. The dynamic pointer structures should bee seen as being more firm than structures that are modified by adding new objects allocated with for example `malloc`.

The language extensions described in this chapter allow the programmer to express *directions*, *coverage*, *disjointness* and *firmness* of pointer based structures. While the first three properties describe a data structure's connectivity and shape properties, i.e. what does the structure look like, the *firmness* property relates to how these properties change over time. For example a pointer linked structure may be constructed during start up of a program, while the program never changes the connectivity or link information; in this case the structure is stable after the initial setup and therefore firm.

While *directions*, *coverage* and *disjointness* will be expressed using data type attributes (using the `__attribute__` syntax from the GCC and Clang compilers), the firmness property is a temporal property, expressed in the control flow using type qualifiers (similar to *const* and *volatile*) and pragmas.

This chapter is organized as follows: Section 4.1 looks into related approaches and their limitations, contrasting these to the Pax C extensions. Section 4.2 gives an overview of the Pax C extensions. Section 4.3 discusses the automatic detection of static pointer structures. Section 4.3.1 goes into detail about dynamic pointer structures. Section 4.4 discusses data restructuring mechanisms enabled by the extensions. Section 4.5 discusses the experiments used to test enabled optimizations. This is followed by Section 4.6 that list the results of the experiments. The experimental evaluation of the language extensions is followed by an analysis of the results in Section 4.7. Finally, conclusions are discussed in Section 4.8.

4.1 Limitations of Other Approaches

Low level approaches aiming to analyze and optimize pointer linked data structures, include for example Lattner and Adve's *Data Structure Analysis* (DSA) [38], automatic pool allocation [36] and the structure splitting work done by others [12, 54]. Basically, in these approaches different properties of the connectivity of data structures and aliasing properties are used to optimize the data structures. The methods are based on compile time analysis, where the compiler determines whether or not different object are aliasing or not. The aliasing and connectivity information is used to optimize the data structures of the program such as automatic pool allocation and structure splitting, or to optimize the control flow when *read after write* dependencies can be resolved.

One of the main limitations of these approaches is that they are too conservative and that they miss optimization opportunities. Another limitation is that the aliasing relationships of more complex expressions can not be determined. As a result of too little (derivable) knowledge at compile time these methods must make conservative assumptions for further optimizations as more aggressive assumptions would result in errors for some cases. For example in the pointer analysis used by the DSA, data structures will be marked as disjoint only if they can be guaranteed to be disjoint. So, it may conservatively identify certain objects as *maybe* aliasing, while they should in-fact be treated as disjoint or as *always aliasing*. This is not to say that the DSA or other algorithms such as Stensgaard [47] are not good at what they do, but they are limited in what they can do as they are supposed to be a normal part of the compiler, and this means that they must not be too slow if the compiler is to provide a reasonable compile time.

DSA for example is context (but not flow) sensitive and able to identify disjoint data structures in a conservative manner. Conservative in this case means that structures marked as disjoint are guaranteed to be disjoint, but structures not identified as disjoint are not guaranteed to be fully connected (i.e. it does not exclude that a structure have disjoint regions). As such, the DSA is conservative albeit very powerful. Although the conservative approach of the DSA does offer the ability to implement automatic pool allocation as mentioned earlier, it is not possible to identify covering traversals of the pool and, for example, apply vectorization or other parallelization methods on the traversals. In fact, any sequential pointer chasing dependency still remain in the optimized data structure and code. The Pax C extensions solves this problem by allowing the programmer to declare covering traversals of data structures. These traversal patterns can then be matched with control flow which in turn allows the compiler to determine when a traversal visits all

elements in a linked data structure. Consequently, if the data structure is pool allocated it will parallelize the traversals, depending on whether there exists additional data dependencies.

The low level approaches are some times combined with runtime mechanisms. In [54] runtime tracing was used to optimize the order of the objects in a pool allocated pointer linked data structure. This method was limited as it was restricted to immutable pointer structures. The runtime tracing in the system had a very high overhead, so it was only used initially until the data structure had been optimized, after which the tracing mechanism was turned off. Consequently it was not possible to take into account dynamically modified pointers. Our extension, does on the other hand offer the opportunity to distinguish between static and dynamic pointer structures and for the latter part, allowing multiple optimization points for the same structure, where each optimization point may use different paths through the structure, or be conditionally executed based on programmer controlled conditions.

Among the high level approaches, several analyses and different language grammars have been developed. In *Shape Types* [17], Fradet and Metayer describe a context-free graph grammar allowing the expression of doubly linked and circular types, which is not possible using traditional type systems. They (like in this chapter) introduced an extension to the C programming language, and supported a kind of checking of the structures on initialization and modification (or *reactions*), based on a special syntax for the modification of the objects. The main limitation in their work was that the shapes where described using a context-free graph grammar that does not blend well with C. Furthermore, modifications of the structures required the use of a special tailored syntax (which allowed the compiler to prove that modifications were valid). So, while their system did extend C, the extensions where not directly compatible with existing C-code. Pax C on the other hand allows for the successive refinement of existing C-code, without any rewriting of the pointer code, and preprocessor directives can easily ensure that attributes are ignored and that the modified code still compiles using a standard C-compiler.

Other work carried out by Hendren et.al. [24, 26], known as *ADDS* and *ASAP* built on extensions to existing "struct" definition syntaxes. They introduced a sense of direction in pointer linked structures (not specifically as a C-extension, but it was aimed at C-like languages), their initial work introduced a way to add directional quantifiers to existing record types. They later extended their system with a more general form of aliasing definitions and descriptions. While being effective in determining aliasing properties of different pointer chains, their approach did suffer from the fact that, as, with the DSA and automatic pool allocation, it is not possible to define coverage of

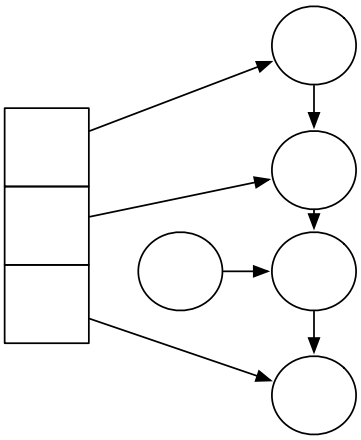


Figure 4.2: Hidden Nodes

pointer linked structures. This is further illustrated by Figure 4.2, where one row is not directly reachable from the array of row pointers. ASAP and ADDS did allow for the distinction between the vertical and horizontal dimensions of such structures, but was not able to describe the absence of the hidden nodes.

In addition to the mentioned language extensions, there has been other high level work such as shape analysis [19]. In shape analysis the compiler analyze the actual shape of a data structure based on for example pointer assignments and allocations. Depending on the system, the resulting shape estimates may differ in both accuracy and correctness. The system was able to reasonably accurately detect the more specific types of structures. While the tree detection could reasonably easily be used for data coverage and disjointness analysis by analyzing the traversal patterns in the code, the more general types of structures detected are difficult to use for this purpose, except for the limited cases of local alias analysis. The disjointness properties introduced by Pax C are stronger than this and can describe global aliasing patterns in a larger structure, based on access patterns with conditions. These disjointness patterns even work on cyclic structures and DAGs.

Another high level approach named PALE [41] was presented by Møller et.al. PALE defined a formal language to describe data structure invariants that could be checked at compile time against a restricted imperative language. The main problem with PALE consist of the fact that execution time and memory requirements of the proofs were too large for practical use. For

example the verification of a red black tree insert took 35 seconds and used 44 MB of memory to check 57 lines of code. While this was the worst test case they provided, the mean and medians both ended up at roughly 30 lines of code, 4 seconds and 8 MB of memory. These results indicate that the computational and space requirements of their methods prohibit a practical application to real world programs that typically consists of millions of lines of code. Even on modern machines, as processor performance has improved considerably since the PALE work was published, the execution time would end up in hours if linearly extrapolated for a 10 million line large program (this is a fair assumption assuming that a program consist of properly encapsulated data structures, all who utilize the PALE logics). In Pax C, we assume a limitation on what can be derived, i.e. more conservative reasoning, and do not let the compiler handle everything. Instead, the attributes in Pax C have been carefully vetted to be verifiable at run-time, and runtime-checks can be emitted if needed.

Note that some systems simply assume that structures are correctly linked. So, a programming error or an invalid object, loaded from a file could potentially break the invariants for the data structure. In the case of PALE this would not matter as it is based on formal proof. However, for Pax C, and also for the ADDS and ASAP, the properties can be violated during runtime. With ADDS and ASAP if the additional data structure properties are not respected by the code, the compiler could in principle generate invalid optimizations that would be very hard to track down. The approach in Pax C is simple, let the program be verified at runtime using a relatively simple model that can catch as many violations as possible when a data structure is used incorrectly.

4.2 Pax C Extensions

Pax C is a set of C extensions forming a strict superset of C, that allow for a more sophisticated description of recursive data structures. Many of the attributes are applied on pointer fields in struct definitions (or on the struct definition itself). There are however a few extensions that are used within function bodies to help the compiler optimize the code. This approach is in stark contrast to the use of other C-extensions that mostly consist of directives embedded in the control flow e.g. OpenMP and Cilk [5].

The first extension of Pax C is the annotation of pointers in data structure types with attributes or axioms that describe either local or global pointer properties. Local pointer properties are the pointer properties that apply to one object and possibly also the object's neighbors; global pointer properties

are those that apply on multiple objects within a data structure. The global properties make use of traversal paths in most cases, these paths describe how a structure is traversed using a regular expression like syntax. The second extension consists of the *static-pointer structure* type qualifier. Essentially, a *static pointer structure* pointer identifies an object where the pointers in that object and in all objects reachable from it are constant. As such, it is known that the shape of the object graph will not change further on in the program. The programmer can define the point from which the data structure's pointers will no longer change in the program by manually adding a type qualifier (similar to the existing C type qualifiers like *restrict*, *volatile* and *const*) and casting a pointer to a non *static-pointer structure* into a *static-pointer* copy of that structure. At this point, the casting operation will trigger a deep copy and restructuring of the data structure, where the traversal pattern is determined from the pointer attributes. In addition to the *static-pointer* qualifier Pax C introduces the notion of *dynamic-pointer structures*. These dynamic pointer structures are similar to the static counterparts, but they allow for the repointing of the recursively typed pointers to other already allocated objects.

4.2.1 Conditional Traversal Patterns

The Pax C extensions consist of a number of pointer attributes, type qualifiers and pragmas. The pragmas and the pointer attributes are used together with *conditional traversal patterns* (CTP). These patterns describe how to traverse a structure.

A pattern consist of a sequence of field dereferences separated by the dot operator (“.”). Between the dot operators, the fields may be expressed as either, field names or using other expressions, such as the | operator that follows each side of the | and the trailing expression (before the next branch). The branch operator is related to the array operator [ ], that does the same thing, but with arrays instead of named fields. The $*$ operator works in similar ways, but expands the expression it is trailing, so it is followed until it reaches null or the starting point when expanding a cyclic field. Note that the $*$ operator only follows single fields or conditional single fields.

The condition operator allows the selection of a path based on a conditional variable/expression, local to the last object in the chain, or a C variable in the case of the pragmas. The conditions are written $\{cond?a : b\}$, where a is used if $cond$ is true and b if it is false. The conditions normally refer to fields in the current object. E.g. $foo.\{bar == 0?left : right\}$ expresses the traversal:

```
current = current->foo;
if (current->bar == 0) {
  current = current->left;
} else {
  current = current->right;
}
```

Finally, the comma operator traverses the full expression to the left of the comma and then the one to the right of the comma. This allows for the restart of the traversal patterns at controlled points.

The expressions are easy to turn into C-code and to understand. The template for conditional traversals has already been shown. For the looping constructs, like $*$ and [ ], the templates are also relatively simple. An array traversal can be emitted using the following code:

```
array = current;
for (int i = 0 ; i < arraylen(array) ; i ++) {
  current = array[i];

  // Rest of expr
}
```

Note that introduced variables (e.g. array in this case) should be uniqued for the traversal. This can be done by appending a serial number to the name, or by embedding it in a scope.

The $*$ works in similar ways, except that the loop is a while loop on the fields. A star traversal can be emitted using the following code:

```
while (current) {
  temp = current;

  // Rest of expr

  current = temp;

  // Traversal (e.g. current = current->next, if
  // the expression is next*)
}
```

For the traversal expressions discussed here, we can insert the visited objects (pre-order) in a sequence, that means that backtracking nodes are excluded from the sequence since they have already been embedded in it and that the successor of a the leaf node in a branched traversal will not be the parent node of the leaf, nor will it be the branching node, but the next unvisited node starting from the branching point. This sequence we will denote

```
typedef struct S {
  size_t bar_len;
  Foo *bar __attribute__((length(bar_len)));
  Foo *baz __attribute__((single));
} S;
```

Figure 4.3: Example of length and single attributes

T_{seq} in the discussion.

In the following section we will give definitions of the attributes that Pax C supports.

4.2.2 Single and Length

The *single* and *length* attributes were defined to work around issues in the C-programming language. In C, pointers to single objects (i.e. references) cannot be distinguished from arrays of objects, and pointers to arrays are not associated with the length of an array. The *single*-attribute forces a pointer to be a reference to one and only one object, while the *length*-attribute allows the specification of an expression that can derive the length, based on data available at the definition point. For pointers embedded in structs, this can refer to a field in the struct. Example of the use of these two attributes are given in Figure 4.3.

Note that these attributes simply associate available length information to the arrays in question. The array length is needed in order to use the array expression for iterating (but not for accessing single objects).

The attributes adds missing information to the C-programming language. Consequently, the property can be used in order to verify during runtime that array accesses do not go out of bounds and that directly assigned blocks from malloc and the related allocation functions are of the assigned sizes. Standard optimizations to bounds checking used by programming languages such as Ada and Java apply to these checks.

4.2.3 Acyclic and Cyclic

The *acyclic* and *cyclic* attributes provide known termination points for common recursive pointer patterns. In effect, as a computer has finite memory, there are three possible shapes for structures linked using one given pointer field: *full cycles*, *straight chains* and *lassos* (see Figure 4.4). Under normal

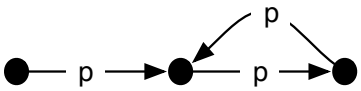


Figure 4.4: Lasso Graph

```
typedef struct Foo {
  struct Foo *bar __attribute__((acyclic));
  struct Foo *baz __attribute__((cyclic));
} Foo;
```

Figure 4.5: Example of acyclic and cyclic attributes

circumstances, the compiler must assume the worst and that a chain may be in any of these formats.

For full cycles and straight chains the termination points for a full traversal of the structure (when p->next == start and p->next == NULL respectively) are easy to derive. However, for lasso shapes this is more difficult.

An example of the usage of the acyclic and cyclic attributes is given in Figure 4.5. In the example, the attributes guarantees that, for all pointers *p*, to objects of type *Foo*, firstly, the traversal `"while (p) p = p->bar;"` will terminate as *bar* is acyclic and therefore must end with a NULL pointer, and secondly, the traversal `"tmp = p; while (p->baz != tmp) p = p->baz;"` will terminate since *baz* is acyclic and the loop terminates at the starting node.

Statements like these can be verified. The violations for the attributes occur if the pointer is acyclic and a full cycle or lasso shape exist, or if the pointer is cyclic and the chain represents a lasso or a NULL terminated chain.

The detection of cycles is a well known problem. One of the more commonly used methods traverses a potential cycle and samples the visited node at increasing intervals. Normally, the distance between the samples increases by a power 2 for every taken sample. This way, cyclic chains will always be detected within two iterations, even if the chain is a lasso. Consequently, iterations along cyclic or acyclic pointers can be instrumented with these checks. If the runtime checks find a cycle before it reaches the starting point of a cyclic pointer, or before it reaches NULL on an acyclic pointer, the program can raise an error. There are ways to avoid these runtime checks if certain conditions hold, though.

Both properties can in some cases be proven by induction. The starting condition is that in the acyclic case, the acyclic field is initialized to NULL,

```
typedef struct Foo {
  struct Foo *bar __attribute__((inverse(baz)));
  struct Foo *baz __attribute__((inverse(bar)));
} Foo;
```

Figure 4.6: Example of Inverse Attributes Use

and in the cyclic case, the field is initialized to point at the same structure it is part of. The compiler can check for this and ensure that after calls to allocation functions (e.g. malloc), the fields are set accordingly before the object is used for anything.

For inserts of individual objects nothing needs to be checked, as long as the initial condition holds for the insertion point and the inserted object. The resulting chain is also null terminated or cyclic.

However, insertion of chain slices is a more complex issue. While, this can simplified in certain condition, it is very complex to do so as the inserted slice may or may not be a single chain, consequently it must be verified that the inserted chain slice is not actually a slice of the destination chain, and this is not always possible. In this case, when not inserting single objects, the checks will be deferred until restructuring time.

4.2.4 Inverse

A pointer field a in an object p is the inverse (or antiparallel) to another pointer b if after following a, b points back at p. The property asserts that as long as p and p->a are not NULL, then p->a->b == p. Common examples include binary linked lists and trees with parent pointers.

Inverse pointers are of interest in order to assist in the verification of cyclicality, and in order to allow for more advanced restructuring opportunities.

An example of the usage of the inverse attribute is given in Figure 4.6. The usage of the attribute in this Figure implies that for all pointers p to objects of type *Foo*: `p->bar->baz == p`. This attribute is assumed to hold at all times, except during transient modifications.

If a code segment modifies an inverse pointer in object p, the inverse must be modified as well to point back to object p before the fields in question are read and used for something eles. This can be checked statically per function. However, if a function does not uphold the attribute, the compiler may emit a warning and make a note that the function made the object p invalid and then proceed with emitting checks at runtime.

```
typedef struct Foo {
  struct Foo *x __attribute__((cyclic));  // Or acyclic
} Foo;

typedef struct Bar {
  struct Foo *a __attribute__((first(x)));
  struct Foo *b __attribute__((last(x)));
} Bar;
```

Figure 4.7: Example of First and Last Attributes

4.2.5 First and Last

The *first* and *last* properties are used to define the starting and ending points of a pointer chain. Depending on whether the chain follows an acyclic or a cyclic pointer field, the attributes have slightly different meanings. By knowing the start and the end of a pointer chain, it is possible to infer that iterations between the start and the end (or the end and the start following inverse pointer) will visit the entire chain. In the cases where the pointer chains can be serialized, this means that it is trivial to determine that the iteration visits every element in the chain. In addition to this, the attributes can help with the verification of the acyclic and cyclic attributes.

Figure 4.7 illustrate the usage of the attribute. In the acyclic case, for all pointers p, pointing out objects of type *Bar*, the attribute *first* implies that when following the field x for all objects q of type *Foo*, there exists no q where `q->x == p->a`. For the *last* attribute on the other hand, if the field b of the type *Bar* has the last attribute, then for all objects p of type *Bar* `p->b->x == NULL`.

In the cyclic case this is a bit different, since there are no logical starting points of a cyclic chain: the first and last attributes are defined with respect to each other. If the pointer p is first, and q is last, with respect to the field x, then `q->a == p`.

For the cyclic case, the verification is obviously trivial at runtime. However, in the acyclic case, it is only easy to verify in the case that the relevant field has an inverse. In this case, if the pointer p has the attribute first with respect to the field a, and field b is the inverse of a, then `p->b` must be NULL. For the case where there is no inverse, the compiler is expected to issue a warning about it. Note that violations can be always be detected using brute force approaches while restructuring, where we could tag visited objects, and which field reached the object.

```
typedef struct Foo {
  struct Foo *parent;
  struct Foo *sibling
    __attribute__((ident(sibling*.parent)));
  struct Foo *child;
} Foo;
```

Figure 4.8: Example of Ident Attribute

4.2.6 Ident

In some cases, different pointers point to the same object in a predictable way. For these cases, the *ident* attribute exist. It can for example be used to infer tree structures in the case where a parent only has a pointer to one of the children who in turn have pointers to it's siblings.

The *ident* attribute use the conditional traversal patterns to identify which objects are identical. As paths cross many objects, the attribute cannot refer to all the visited objects, instead it refers to the leaf points in the traversal paths. An example of the *ident* attribute is given in Figure 4.8. In the figure, the sibling nodes parents will all point out the same parent.

4.2.7 Covering and Disjoint

The *covering* and *disjoint* attributes utilizes the conditional access patterns in order to describe whether or not the traversals will visit all objects or whether the traversal will be completely disjoint.

A conditional access pattern will traverse multiple objects and the sequence of objects, known as T_{seq}, may visit objects several times in some cases (e.g. in case there are multiple paths to the same object).

For the *covering* attribute, the following is assumed to hold: for all objects in T_{seq}, there are no paths from any of these objects to an object of the same type, unless those objects exist in T_{seq}. Note that, in some cases, there are several pointers, all pointing to different covering sets (e.g. two different sparse matrices), for this reason the attribute has a second argument, a set id. Thus, the attribute applies only to the objects identified with the set id.

For the *disjoint* attribute, the definition is simpler: for a disjoint CTP the resulting T_{seq} is a set, or simply, every object in T_{seq} occurs only once.

Verification of disjoint and covering is deferred until restructuring. Where it is reasonably trivial. If we restructure using some path. The path can be checked reasonably easily by tagging visited objects, if the path matches the

```
typedef struct Foo {
  struct Foo *x __attribute__((acyclic));
} Foo;

typedef struct Bar {
  struct Foo *a[] __attribute__((covering(a[].x*, 0)))
                  __attribute__((disjoint(a[].x*)));
} Bar;
```

Figure 4.9: Example of Covering and Disjoint Attributes Usage

covering or disjoint paths. Traversals along disjoint paths may never reach an already tagged object, and traversals along covering paths may never visit objects that have pointers to objects not in the path. As will be seen in Section 4.4 the restructuring operations, must rewrite all the pointers to moved objects. The remapping is done in a second pass after the objects have been moved. In this case, if a pointer is not found to be remapped to a new location, then the sequence cannot be stemming from a covering path. However, the operation may be costly, so like with most runtime checks, it should be controllable by compiler switches.

4.2.8 Static Pointer Structures

Pax C makes use of a notion we call *static-pointer structures* that can be applied on certain data structures. In essence, *static-pointer structures* have constant pointers. The formal definition is a bit more complicated: *For a static-pointer structure object, the object is used in a type safe manner; all pointers in that object that point to aggregate objects that in turn contain pointers, are immutable and point at static-pointer structure objects.*

It is arguably very difficult for a compiler to determine which objects are *static-pointer structures* and where in the programs they are so. In practice it is also impossible to do so in many cases, especially when the compiler has an incomplete view of the program. Therefore we introduce a type attribute that can be attached to structures in the same way the structure can be declared *constant*, *volatile* or *restricted*. It is up to the programmer to add casts at relevant points in the code that declare to the compiler that a data structure is a *static-pointer structure*. This information can be used by the compiler to generate data structure transformations.

As mentioned in the introduction, the automatic detection of *static-pointer*

```
Foo *bar = makefoo();
staticptr Foo *baz = (staticptr Foo*)bar;
```

Figure 4.10: Static-Pointer Casting

structures is conservative in nature. The automatic detection will help the programmer to incrementally add attributes to the code. A method able to detect *static-pointer* structures is described in Section 4.3.

4.3 Conservative Static Pointer Detection

In some cases, it may not be trivial for the programmer to determine that a data structure is a *static-pointer structure*. In the case that the programmer has only been setting the pointer attributes in the data structures, it is possible for the compiler to detect naturally occurring *static-pointer structures*. In order to do this, it is possible to employ an extended version of the DSA (the full details of the DSA being out of the scope of this chapter, but it is briefly described in Chapters 1 and 2). The modifications add per-field flags and an additional top-down pass, neither of which should be resulting in any major performance hits due to their low complexity.

The DSA employs a local analysis, followed by a bottom up and top down analysis of the call graph, where local analysis information is passed upwards (i.e. due to functions returning values or modifying objects passed into the functions), and merged (when a pointer variable has been used or assigned in multiple branches) and then passed downwards in the call tree.

Note that the local analysis followed by bottom up and top down passes can easily be used to propagate information about verified attributes. When nodes are merged (because they are defined in different branches), the validity flag for the attribute is simply "bitwise anded" together, meaning that if one branch results in an invalid attribute then the merged representation is invalid as well. This is done after the local analysis has determined whether the attributes are valid, assuming that the preconditions are valid.

In order to detect static pointers, one need to enable per-field mod / ref flags, allowing the compiler to detect in which contexts the pointers within a data structure are written to. The second addition is to add two additional per-field flags. These are the *free* and the *nullification* flags. The free flag is set if a field escapes to the free function and the nullification flag is set if a pointer field is written the constant value NULL (in this case, the mod flag is not set). When this is done and the local phase of the DSA has been completed we do a

Figure 4.11: Static-Pointer Structure Detection. Flags are: R = read, N = nullify, F = free.

bottom-up type merge (where all DSA node types are merged) and a top-down traversal (where the same is done). It is then possible to do another top-down traversal (starting with the main function). During this traversal, we build a per-data structure inter-procedural dependence graph where the nodes that create the data structure will be dominating the successors (i.e. they will be executed earlier in the program flow). This step is context sensitive like the DSA and not flow sensitive[4]. In this dependence graph we are looking for a pattern where the first flags are modifying the pointers and the rest of the graph is not modifying the pointers (except for the last couple of nodes where the pointers may be nullified). The pattern that is searched for is illustrated in Figure 4.11.

After detecting a *static-pointer structure*, the compiler can insert a *static-pointer* cast just before the *static-pointer* region (see the middle *usage* node in Figure 4.11). This cast is inserted just before the usage phase. For the tear-down phase, the original data structure (not the converted one) should be passed along, this is safe since the structure in question is no longer used. At this point it is useful to insert a runtime call to release the *static-pointer structure*.

4.3.1 Dynamic-Pointer Structures

In contrast to a *static-pointer structure,* a *dynamic-pointer structure* allow the pointers to be changed and objects relinked, however it does not allow for new objects to be allocated and inserted in the pointer linked structures (the programmer can overcome this limitation by using fixed sized pools of objects). Casting a normal pointer to a *dynamic-pointer structure* will otherwise work similar to the *static-pointer structures*.

The dynamic-pointer structure was added to the Pax C extensions in order

[4]For a summary of the terms context and flow sensitive, see [25]

to be able to handle problems where an iterative solver is modifying the pointer structures slowly converging on the final result. Since, the *dynamic-pointer structures* allow the pointers to be modified they are more flexible for the programmer than the *static-pointer structures* defined in the previous section. On the other hand, they do not allow for certain optimizations that rely on having a known iteration space.

4.4 Restructuring

When an object is casted into a *static-pointer* copy, the compiler is free to reorganize and restructure the pointers in the data structure. Although, some restructuring can possibly be done before this (such as some forms of pointer compression, and other type modifications), the *static-pointer* property does allow for more options.

A type-specific automatic restructuring function can be generated by the compiler when it knows the attributes mentioned in section 4.2 and calls to this restructuring function at the locations where a normal object is casted into a *static-pointer* copy. The main premise for generating such a function is to be able to determine a set of covering (and preferably disjoint) paths in the data structure.

Naively, it is always possible to determine a covering set by a DFS traversal of the data structure. However, this is not practical for several reasons. For example, a DFS will in many cases not correspond to the natural iteration order of a more complex data structure and it will have to deal with a lot of bookkeeping information to avoid endless recursion in the presence of cycles.

The rewrite operation does two things, firstly it transforms the pointer types, and secondly it does a deep copy of the rewritten data structure (following the covering path) into a new single data block (an array). The pointers are converted in order to refer other objects within the array using indices. Consequently, when converting a pointer-linked data structure into the *static-pointer* version, there are three forms of pointers to consider: indices within the new data block, fat pointers that refer to another data block and slices that refer to a subset of objects within a data block. An index is used when there is a self recursive pointer within the same set. A fat pointer, which is a tuple of an index and an array pointer, is used as pointers to single objects in some other set (or data block). A slice is a triple of a start index, end index and an array pointer. Slices are used when pointing out the first object in a known subset (if this is in the same array as the current object, it can be shortened into just the start and the end index), for example, when we point

Original Type	Introduced Types

struct A {A *a; B *b;}	struct A_1 {
	int a; B_fatptr b; }
	struct A_array {
	int len; A_1 elems[]; }
	struct A_fatptr {
	int idx; A_array *arr; }
	struct A_slice {
	int start; int length; A_array *arr; }

Table 4.1: Type Conversion. The ∘ is used as a generic operator, e.g. == or !=.

Original	Translated
*p	p.arr->elems[p.idx]
p∘NULL	p.idx∘0
q=p->R	q.idx=p.arr->elems[p.idx].R q.arr = p.arr

Table 4.2: Statement Conversion

out the first object in a list like structure such as a row in a sparse matrix. The introduced types are illustrated in Figure 4.1.

Since the types are modified, the addressing code needs to be rewritten. Essentially all pointers to *static-pointer* objects will be turned into fat pointers. This includes pointers passed as arguments to functions. Initially only three translations need to be done in the code when accessing the fat pointers. We show this in C syntax in Figure 4.2.

This will not necessarily result in any improvements of the performance of the code in question. Several additional optimizations are performed on top of this. Most important is the rewriting of pointer-chasing loops into index-incremented loops (see Figure 4.3). This is possible if the pointer chasing goes in the same direction as the covering and disjoint paths and the traversal covers a full path or the full range between two known elements, such as a full structure or a slice.

Original	Translated
while (p) { p = p->R; }	for (; p.idx ¡ p.arr->len ; p.idx ++)

Table 4.3: While Loop Conversion

Note that depending on whether or not the initial data structure has been pool allocated or not, the rewriting mechanism can either use an indirection array or a hash-table to store the old to new pointer mapping. The rewrite is a two pass affair, where in the first pass the data structure is copied over to the new data block, and in the second, the pointers are rewritten to point at the new data locations. The only pointers that can be updated on the first pass are those that point at the "next" object.

4.5 Experiments

4.5.1 Sparse Lib

The SPARSE library (see [32, 33]) utilizes orthogonally linked lists to represent sparse matrices. The library provides a matrix frame type describing the matrix structure (i.e. the size, starting points of rows and columns) and pointer-linked matrix elements that encapsulate a non-zero element. The data structures are well suited for optimization using the Pax C extensions. The updated types in the (not showing irrelevant data fields) is listed in the following code:

```
struct MatrixElement {
  double    Real;
  struct MatrixElement  *NextInRow
    __attribute__((single))
    __attribute__((acyclic));
  int           Col;
  int           Row;

  struct MatrixElement  *NextInCol
    __attribute__((single))
    __attribute__((acyclic));
};

struct  MatrixFrame {
  int NrOfElements __attribute__((set_size(1)));
  struct  MatrixElement **FirstInCol
    __attribute__((covering("[*].NextInCol*", 1)))
    __attribute__((disjoint("[*].NextInCol*")));
  struct MatrixElement     **FirstInRow
    __attribute__((length(Size)))
    __attribute__((covering("[*].NextInRow*", 1)))
    __attribute__((disjoint("[*].NextInRow*")));
  struct MatrixElement     **Diag
    __attribute__((length(Size)))
    __attribute__((in_set(1)));
  int Size;
};
```

In the code listed above, the covering attributes not only specify the paths through the matrix, but also the set identifier. The set identifier is needed in case the matrix frame would point out two different disjoint matrices. The definition of disjoint rows and columns allow the compiler to not only restructure the data, but also to expose the inherent parallelism in a loop over the rows. In the matrix type, it is not possible to infer parallelism automatically in the general case as the code may modify the matrix contents during iterations[5].

We carried out experiments using a number of kernels based on the SPARSE library data types, namely *dsolve*, *jacit*, *pcg*, *spmatmat* and *spmatvec*. These kernels are part of the SPARK00 [51] benchmark suite. Most of the kernels traverses the matrices in either row or column order. There is one notable exception, *dsolve*, that includes the traversal of an LU-factorized matrix. In this case the matrix is traversed partially in row-wise order and partly in column-wise order. Figure 4.12 shows the average speedup of the different kernels when the matrix data is read using different memory orders. For row-wise ordered data, the restructuring is not helping. This is due to the fact that matrix is already perfectly ordered in memory. For the column-wise order, the data is is not ordered in memory in the same way as the traversals, therefore we can see an improvement in performance here. For randomly ordered data, the data ends up being well ordered after the restructuring, so the improvement

[5]Naturally, a loop not modifying the matrix contents can have its row or column traversals executed in parallel.

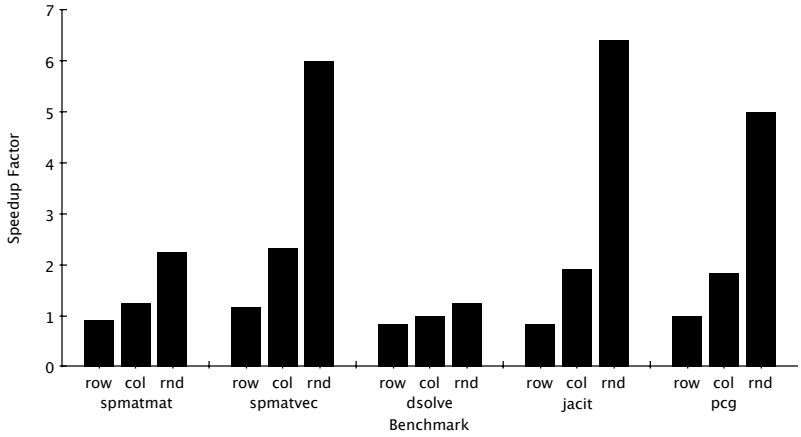


Figure 4.12: SPARSE lib kernels speedup factors grouped by benchmark.

is considerable.

4.5.2 MCF

MCF is known as *181.mcf* in the *SPEC2000* (or *429.mcf* in *SPEC2006*) benchmarks. Previous work where MCF has been targeted for optimization includes among others [20] where the MCF data structures were subjected to *peeling*, *splitting* and *field reordering*. These transformations are not global in the sense that they consider the connectivity of the objects, rather than transform individual objects into a different layout.

MCF is implemented using a structure containing an array of nodes and an array of arcs (each node also has a number of implicit arcs relating to the spanning tree that is being computed). The main data structure is traversed multiple times with different (potentially) conflicting access patterns. For example, the function `update_potential` traverses the nodes following the spanning tree using a DFS pattern, however, other functions iterate over the node array and traverses chains of nodes if certain conditions are met. In fact, reordering the nodes after the DFS traversal will slow down MCF in total (although the `update_potential` function itself runs substantially faster).

In our experiments we show that we have been able to achieve restructuring in MCF without the use of tracing, on which the restructuring systems to this day have been relying on (e.g. [54]). MCF is an iterative solver, as such the

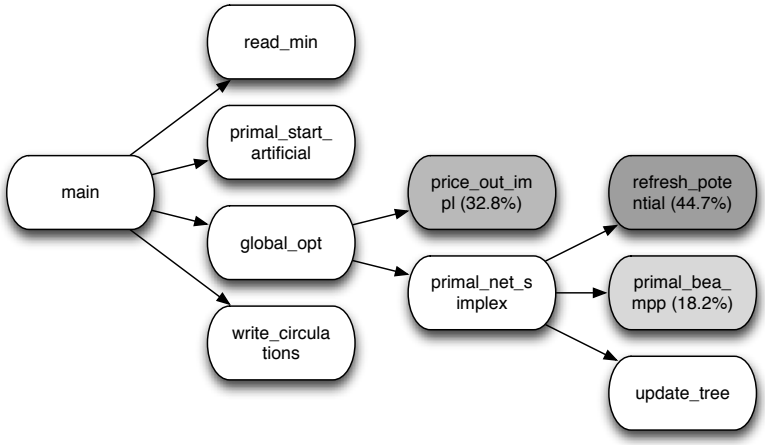


Figure 4.13: Simplified MCF call graph with execution times in % for the reference data set.

underlying graph based data structure is continuously changing while the data structure is being optimized.

As is well known, to determine how to optimize a program one needs to know the hotspots where the program is spending most of its time. We used profiling to determine that in MCF, the execution time was primarily centered around three functions. These functions are `refresh_potential`, `price_out_impl` and `primal_bea_mpp`. For the reference data set, the functions were using 96 % of the program's execution time. The simplified call graph in figure 4.13 illustrates this further. As can be seen, any optimization for the code must target these functions or seriously suffer from diminishing returns.

Obviously our first attempt was to optimize the access pattern for the `refresh_potential` function that was the most costly of the three. The optimization centered around the fact that MCF only modifies a small part of the data structure as the program gradually tries to build up the solution. So, essentially by adding the restructuring pragma to the beginning of the *refresh potential* function, using the access pattern used in the function, it was possible to carry out the restructuring step.

Essentially, the *refresh potential* access pattern is a DFS. This DFS is described by the attributes that can be set on the MCF data structures. By stat-

ing that there is a covering and disjoint path in the `network_t` structure type using the following expression: `covering(nodes[0].(child|sibling)*)`, The permutation vector is then generated following this traversal and the nodes are reordered.

This optimization however caused slowdowns in the other hotspots and although the `refresh_potential` function ran substantially faster (40% including overhead from the data restructuring), the new access pattern slowed down the program as a whole.

As the *refresh potential* function did result in slowdowns, the other hotspots had to be reconsidered. The `primal_bea_mpp` function was determined to use a much too complicated access pattern for both manual and automatic analysis. Instead the `price_out_impl` function was considered for optimization. The function essentially checks every third arc and if a tag value is not set to a constant value named FIXED, it traverses the nodes for the arc backwards (i.e. from the tail and backwards). The access pattern can be codified in the following way: `{net.arcs[i=1..$length, step 3].ident != FIXED} ? net.arcs[i].(tail.mark)*`. By using the pattern given here, we were able to speed up MCF as a whole compared to the non (access-pattern) optimized version. The steps needed are to ensure that the *length* and *covering* properties of the network type are known, then the restructuring pragma needs to be added to the code.

```
typedef struct {
  node_t *nodes
    __attribute__((length(stop_nodes-nodes)))
    __attribute__((covering(nodes[0 .. $length])))
    __attribute__((covering(nodes[0].(child|sibling)*)))
    __attribute__((disjoint(nodes[0].(child|sibling)*)))
      ;
  node_t *stop_nodes;
  arc_t *arcs __attribute__(length(stop_arcs-arcs));
  arc_t *stop_arcs;
} network_t;

network_t *net;

#pragma restruct (node_t) net using\
  {net.arcs[i=1..$length, step 3].ident != FIXED} ?\
  net.arcs[i].(tail.mark)*
```

Note that the mark field is not an arc pointer but a pointer sized integer. However, the field is used as a pointer in the code using type-unsafe casting and the benchmark had to be modified by changing the type of the mark field to the proper pointer type. In the future, Pax C may be extended with casting operations in the CTP syntax.

4.5.3 Parallelizing Refresh Potential

Refresh potential traverses the entire spanning tree of the graph of nodes. The only intra node dependencies that exist are from the parent nodes. As such, the function is suitable for parallelization using the traditional *divide and conquer* approach. A complication for this is that the trees are not balanced, meaning that additional steps need to be taken to properly parallelize the function.

We show how parallel execution of the refresh potential function can be implemented with minor changes to the code. The first step is to assign the covering and disjoint attributes to the MCF data structure as described previously. The second step is to rewrite parts of the `refresh_potential` function using a recursive traversal instead of the current pointer chasing one. When testing this with the large reference data set from MCF, we saw that a recursive version did not result in any performance differences compared to the standard iterative traversal. In this case, rewriting the code using recursion makes it easier to analyze the function.

The function can then be parallelized by invoking some of the recursive calls in parallel and waiting for the call to finish before returning from the caller. This is similar to how the *cilk [5]* programming language handles parallelism. Our manual attempts with the *cilk* language, however, did not succeed due to the heavy overhead. On the other hand the *cilk* version served as a conceptual proof. A two thread *pthreads* based solution was successful in speeding up the code.

Essentially, the core part of the refresh potential function is rewritten as illustrated in Figure 4.14. As seen, the *disjoint* attribute is embedded in the *node_t* type. Without this addition, the compiler would have to assume that the subtrees were fully aliased and would not be able to proceed with the parallelization attempts.

The compiler now has to analyze the recursive calls of the function, these are in the direction of the sibling and the child pointers. Each function also accesses the predecessor. Table 4.4 illustrates the dependencies for the refresh potential function extended one step. The simplified dependencies show the actual dependency after taking into account the *ident* and *inverse* attributes. The calls to refresh_pot have the traversal patterns: a) *sibling.(sibling|child)*$*$

```
typedef struct node_t {
  struct node_t *predecessor
    __attribute__((single))
    __attribute__((ident(sibling*.predecessor)));
  struct node_t *child
    __attribute__((single))
    __attribute__((inverse(predecessor)));
  struct node_t *sibling
    __attribute__((single));
} node_t
  __attribute__((disjoint(
    predecessor.(predecessor|sibling)*,
    ((child|sibling)*) )))
  __attribute__((disjoint(
    (child.(sibling|child)*),
    (sibling.(sibling|child)*) )));

int refresh_pot(node_t *node) {
  int a, b = 0, c;
  if (!node) return 0;

  a = refresh_pot(node->sibling);
  if(node->orientation == UP) {
    node->potential = node->basic_arc->cost
                    + node->pred->potential;
  } else {
    node->potential = node->pred->potential
                    - node->basic_arc->cost;
    b ++;
  }
  c = refresh_pot(node->child);
  return a + b + c;
}
```

Figure 4.14: Rewritten Refresh Potential

Dependee	Dependency	Simplified
node->pot	node->pred->pot	-
node->sibling->pot	node->sibling->pred->pot	node->pred->pot
node->child->pot	node->child->pred->pot	node->pot

Table 4.4: Dependencies for refresh potential

and b) *child.(sibling|child)∗*. If the two patterns are independent (i.e. disjoint), and do not access other data except from reading, then clearly the function calls can be called in parallel. In this case these patterns match the programmer-specified attributes exactly. The traversals are thus disjoint. The additional access in the function using the predecessor node is also clearly not interfering with the traversals, which can be seen from the other disjoint attribute. The only direct dependency is the *child* node's *potential* field which depends on the current node's *potential* field. Therefore, it is possible to conclude that the first recursive call on the sibling may be executed in parallel with the remainder of the function.

When it is known that the traversals are disjoint, the compiler needs to figure out how often these parallel calls should be done. Preferably, all the calls taking the *sibling* pointer would be parallel. Unfortunately this does in most environments generate a substantial overhead. Our solution to this was to utilize the mechanisms in the restructuring system and add a periodic count of the subtree sizes for each node. The function was then cloned and the entry point modified to check whether the subtrees (reached by traversing the child and the sibling pointers) were of sufficient sizes. If both subtrees are of different sizes, the sibling node is handed away to a worker thread which the current thread waits for before using the result from the function call.

Note that only one branching point was used. The main drawback of this is that all parallelism will not be exploited, however, as will be shown later in the experimental results, it still yielded decent results.

4.6 Results

Our experiments were carried out on a number of different machines with somewhat varying results. Three major experiments were carried out during the exploration of the MCF code, Firstly, we investigated optimization of the `refresh_potential` function only. Secondly, we investigated an optimization focusing on the `price_out_impl` function. The code was passed through the -

O3 optimization flag of the *clang* compiler to ensure that we were not repeating standard built in optimizations from the compiler. Thirdly, we focused on parallelization speedups and the `refresh_potential` function was parallelized with a single branching point taking into account the subtree sizes for the *sibling* and the *child* pointers.

The *refresh potential* test was executed by comparing a base version without restructuring, with variants that trigger restructuring every nth call to the function. We tested several intervals for the restructuring. Another test also compared two different versions, one generic using a very fast hash table for mapping old pointers to new pointer values (which is needed when objects are not allocated in arrays) and a second version using an index based remapping vector. The index based remapping vector is applicable on many different data structures, and should be used whenever possible. For this experiment we used a small data set[6].

The *price out* optimizations were tested using a hybrid access pattern method. Essentially two access patterns were used, the first one being the covering pattern from *refresh potential* and the pointer attributes; the second one being the pattern used in the *price out impl* function. The two access patterns were interleaved and tested with different combinations (for example 2 nodes using one pattern followed by 4 nodes using the other pattern). For this experiment we used a large data set[7]. Since one of the patterns was not covering, the traversal of this non covering pattern had to be followed by the covering pattern so that all nodes could be moved into the new data block.

The parallel version of `refresh_potential` was tested on an x86 machine running Linux. The experiment utilized pthreads to handle the parallel work in a fork-join like way (emulated using mutexes to improve performance). Although a single branching point is not optimal from a parallelization point of view, it serves here to show that even a simple mechanism like this can have a substantial effect. The refresh potential test utilized the large reference data set from the MCF code.

4.6.1 Refresh Potential Optimizations

The results for the refresh potential function optimizations are shown in table 4.5. The table shows how often restructuring was done, the time for the refresh potential function, the overhead from restructuring and speedup of the function. Note that the program's total execution time went up in all cases

[6]Data set named *train* distributed with mcf.
[7]Data set named *ref* distributed with mcf.

Test Case	Base (s)	Overhead (s)	Speedup (%)	Speedup Idx (%)
Base	1.55	N/A	N/A	N/A
100	0.9	1.58	-37.5	46.5
200	1.1	0.8	-18.4	31.4
400	1.2	0.43	-4.9	24.7
800	1.25	0.21	6.2	22.0
1000	1.27	0.17	7.6	20.4
1100	1.31	0.14	6.9	17.1

Table 4.5: Optimizations targeting refresh potential

Machine	Serial	Parallel	Speedup (tot ; func)
Core 2, 2.66GHz	130.3 s	122.4 s	6.1 % ; 13.6 %

Table 4.6: Parallelization of refresh potential. Total execution time.

(this is not shown in the table) as the restructuring caused other functions to run slower as they used different access patterns, the improvements in the tables are solely for the execution time of the `refresh_potential` function. The overhead column shows the overhead from the restructuring when hash tables were used to keep track of pointers. The indexing version was 10 times faster at restructuring than the hash table version (only the derived speedup is shown in the table).

4.6.2 Price Out Impl Optimizations

The graphs in figures 4.15, 4.16 and 4.17 show the speedup obtained on different platforms using the optimized MCF data structures. The different horizontal axes indicate the interleaving levels for the two different access patterns. The primary pattern is the covering access pattern and the secondary access pattern is the one responsible for optimization (i.e. the pattern used in *price out impl*). The two axes together mesh out the speedup on the vertical axis.

4.6.3 Parallelized Refresh Potential

The parallelization tests where run on a quad 2.66 GHz, Core 2 machine. Table 4.6 shows the average execution time during 20 runs on this machine. The speedup is reported for whole program and the `refresh_potential` function.

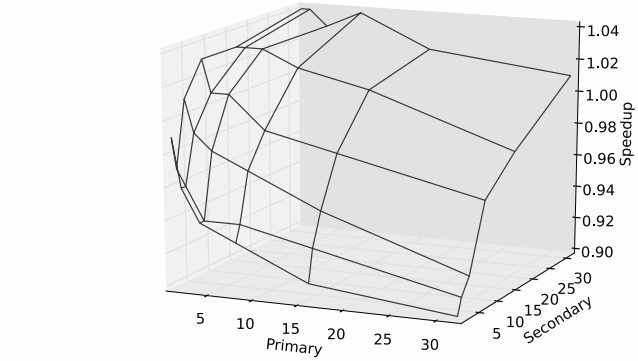


Figure 4.15: Core 2 @ 2.66 GHz, 4 GiB RAM, 6 MiB L2, Linux

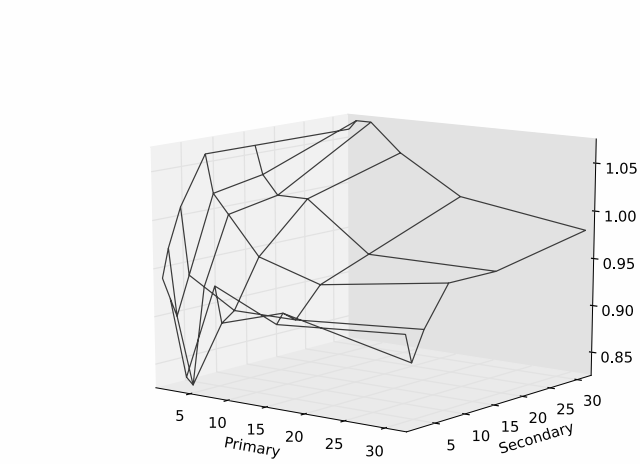


Figure 4.16: i7 @ 2 GHz, 4 GiB RAM, 6 MiB L2, OS X Lion

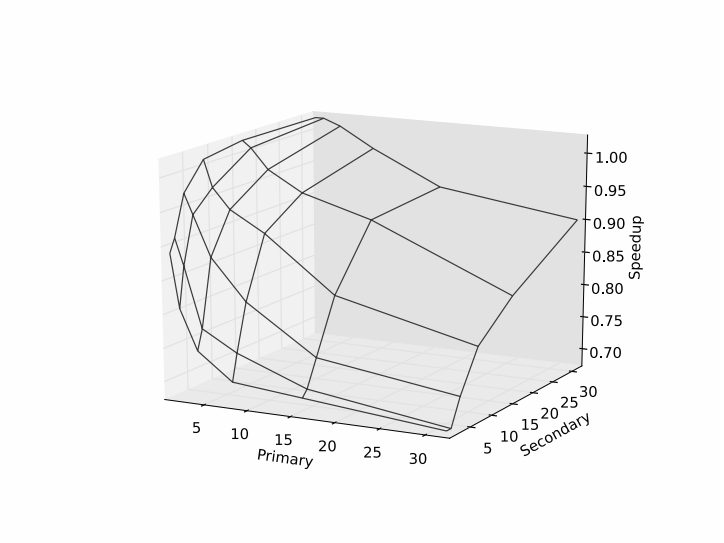


Figure 4.17: i5 @ 3.20 GHz, 3.8 GiB RAM, 4 MiB L2, Linux

4.7 Discussion

For the initial experimentation with the *refresh potential* function we see that when using index based vectors the optimization quickly pays off (locally). Although the total execution time was higher for the program, the approach is still valid as many applications will not show the same behavior. In fact, the experiment confirms that substantial speedups of certain types of applications are feasible with the approach laid out in this chapter.

For the second experiment with the *price out impl* access pattern, the optimization experiment demonstrated that the approach is feasible, has a total performance improvement on the application and is applicable to more complex programs. This improvement was seen on all the platforms we tested. Although this improvement was very small on the *Core i5* which we guess is due to the limited amount of cache in the processor. However, this should have been expected as the reordering is primarily a cache optimization of the data. For the experiment in particular, only the secondary access pattern should be used, though the primary pattern is still necessary to make a complete restructuring.

We also showed how a minor rewrite of the *refresh potential* function could

enable semi-automatic parallelization of the function. This is substantial in itself from a conceptual level, and even as the performance was improved (by 6.1 % for the whole program and 13.6 % for the function by itself). The method used only one extra worker thread and additional parallelism remains unexposed. While there are programming models that allow the programmer to express recursive parallelism such as the mentioned *cilk*, for many similar codes the runtime overhead would simply be too large. The Pax C extensions enable similar optimization in an entire program by concentrating the modifications on a few data types instead of in the code. Better hardware support such as proposed by the microgrid team [6], could possibly be applied to relieve the situation for these codes.

4.8 Conclusion

In this chapter we presented extensions to the C programming language that enable the restructuring of large pointer-linked data structures. The restructuring step converts all pointers into either fat pointers, slices or indices. This rewrite of data structures enables position independence for pointer-linked data structures, and the reordering of objects into a more optimal memory layout. It is feasible to apply compile-time checking of the validity of the pointer attributes in many cases, and, when not, it is possible to resort to runtime checks.

The restructuring done with the help of the language was demonstrated to give noticeable performance improvements for some common problems involving sparse matrices and spanning tree algorithms, and this without the utilization of the additional aliasing information that the attributes make available to the compiler.

In general it would be nice to express any kind of property using the most generic language possible. However, while it is of interest to have a general programming model for describing data structures, the fact is that there is a significant tradeoff between expressiveness and programmability. A language extension must be usable for a human programmer. There is thus a tradeoff between a purely theoretical model and a more practical approach that is easy to learn and work with. For this purpose, we have tried to avoid defining a too generic system that is difficult to use, and a too verbose and extensive system that is difficult to learn. We aimed at a set of attributes and language extensions that is minimal but still functional, and although the balance between ease of use and generality is very much subjective and is difficult to quantify in a testable way, we believe that by keeping the set of attributes minimal

but still functional the Pax C extensions described in this chapter manages to strike a good balance between ease of use and generality.

Chapter 5

Hardware Based Restructuring

In Chapter 2 we presented a method based on compiler analysis in combination with runtime tracing and object reordering. This method was based on the compile-time identification of type-consistent collections of objects who could be automatically pool-allocated by the compiler. Whenever these objects where used repeatedly, the compiler emits tracing code for the first pass and then applies the restructuring operation on the linked data structure. The method resulted in substantial speedups on several matrix-based benchmarks.

A more optimal restructuring of linked data structures resulted when logically adjacent objects where placed in a corresponding memory order (an object a is logically adjacent to object b, if there is a pointer from object a to b).

An immediate difficulty with objects being logically adjacent and restructuring is that in any kind of linked recursively typed data structure, except for the most simple case of a singly-linked list, an object will have multiple logically adjacent objects and therefore there may be multiple paths possible through the data structure.

When a set of objects is restructured, the objects are reordered in a compressed chain, that in many cases can be seen as an array. This reordering is called chaining, and the chained objects are called a chain.

In this chapter we introduce a method for applying restructuring by means of hardware support. Other groups, have explored hardware based approaches in order to solve the linked data structure problem. In [9], the additions of

pointer caches to speed up irregular pointer accesses was explained. This approach is very similar to the prefetch stream detectors. Automatic prefetching systems in processors tend to detect patterns that are regular, which works very well for arrays, but not well for pointer based structures that are spread out in memory. The pointer cache on the other hand works by specifically caching the addresses of pointers (on the heap) and their target addresses, this information can then be used in order to prefetch the next object. The approach uses processor state to predict the pointers to fetch. Other groups have explored stateless prefetching of pointer chains. In [31], a directive-based prefetching system is described that allows the compiler or programmer to emit prefetching directives that are able to describe both strided and chained accesses. The system also allows for the fetching of parallel chains independently. Both of these approaches differ from our approach substantially, primarily in that we actually serialize the data.

Our system works by the programmer or compiler declaring some objects as chainable. This is done by ensuring there is space in the object for an additional metadata field maintained by hardware and that the addresses of the objects are easily detectable as chainable object addresses. These objects are serialized into a dedicated chain memory. For each chainable object that is serialized, the original is kept but modified with a forwarding address that indicates to which chain the object belongs and where in the chain the copy of the object is located. Accesses through the virtual addresses can thus be redirected to the chain memory by inspecting the object's metadata field that contains the actual location of the serialized representation. It is obviously not practical to carry out these lookups, so an additional system that tracks stack pointers and their corresponding chain memory location is introduced. In order to explore the potential of this method, this chapter provides a simulation-based assessment of the cost vs performance improvements of the method.

Note: while this chapter does not work out the details of an optimal chaining mechanism, it does present the basic mechanism by which objects are chained using only one of the pointers.

5.1 Implementation

One of the problems with restructuring and chaining is determining that an object is actually chainable. For our purposes, we have decided to work with specially dedicated *chainable memory*. The *chainable memory* is essentially normal memory that has been declared as chainable by setting a special bit accessible to the processor. Objects that have been chained in turn are stored

in *chain memory*.

If hardware is supposed to chain objects, a pair of object parameters are needed. The first is the object layout that defines which fields in an object are recursively typed pointers. The other parameter is the size of the object. It may be possible to determine these at runtime, however we assume that they are explicitly registered with the processor by the program.

Programs use virtual addresses in order to point to variables in memory. These addresses do not change, and we ensure that they are translated into different physical addresses if the objects are chained. The translated addresses would normally be cached, and it may be impractical to build large content addressable memories that are sufficiently fast and energy efficient. Instead of caching the translation for the addresses, we can reduce the number of translation entries by storing the translations of the pointers keeping chainable addresses instead of the direct address translations. These pointers we call active pointers.

Note that even though the active pointers are available, whenever a program deviates from the assumed path with which the chain was built, the active pointer cannot be used to lookup the chain address. In this case, the address lookup goes through normal memory, where we explicitly embed a forwarding pointer in each object to describe the location of the object in *chain memory*. This lookup is similar to a VM table walk, except there is one translation per object, and not per page.

There are three fundamental aspects to the system: detecting chainable pointers, tracking the active base pointers and chaining objects. Each of these is discussed in the following sections.

5.1.1 Detecting Chainable Objects

An address must be identifiable as chainable. We explore two ways this can be accomplished. The first discusses the use of the virtual address format of the processor, and the second discusses the use of the virtual memory page tables. Both methods are applicable, but have different drawbacks and advantages depending on the architecture of the system. Note, however, that there is essentially no performance difference between the methods. In both systems, the programs need to be modified to utilize these addresses. These modifications can be focused on the code that allocates the data structures that need to be returning chainable addresses instead of normal pointers.

Dedicating Bits

One way of identifying chainable objects is to dedicate a bit in the virtual address space. This method is relatively simple to implement and doing so essentially reserves half of the virtual address space. However, this may in turn have substantial backward compatibility problems for existing software and operating systems. For example if the system is implemented on an existing architecture it is likely that linker scripts would have to be rewritten and a large amount of the software recompiled, though several methods can be used in order to avoid the recompiling of all software.

One way of avoiding recompilation is to use methods common in embedded systems, where processes declare whether or not they want to use the floating point unit. By doing so the operating system can optimize the context switching overhead. Similarly, it is possible to signal to the processor that a running process is compiled for the chaining engine.

An advantage of a dedicated bit is that it is very easy to construct efficient logic that detects chainable addresses (see Figure 5.1). However, one issue for such an approach is that it is difficult to divide the chainable memory into anything but power-of-2 sized regions. Subdividing the dedicated address space is necessary if we want to have multiple subregions (for example if we have different types of linked objects). While it is easy to build up a tree of dedicated address prefixes that describe a different chainable address region, it may be of interest to the programmer to have a more efficient utilization of the address space.

As mentioned, dedicating a bit, presumably the most significant bit of an address, as a chainable memory prefix bit, may have implications for software on existing systems. On the x86-64, we can solve this using the non-canonical addresses (see [28]) for chainable memory. Note that it is not possible to dedicate the *Least Significant Bit* (LSb), even though there are alignment restrictions on the objects and the fields in the objects. The problem with doing this is that there is no way to distinguish a load to a generic byte string (using an alignment of 1) from a chainable memory load, this is further illustrated in Figure 5.2.

The x86-64 is divided in canonical and non-canonical address ranges, and while the architecture itself supports the use of the full 64-bit address space, no implementation presently does this. The canonical addresses, that are the normal addresses that a program uses are further subdivided in a high and a low half. The non-canonical addresses are the unusable addresses between the two canonical address ranges and the processor will generate exceptions if a program access the non-canonical address space.

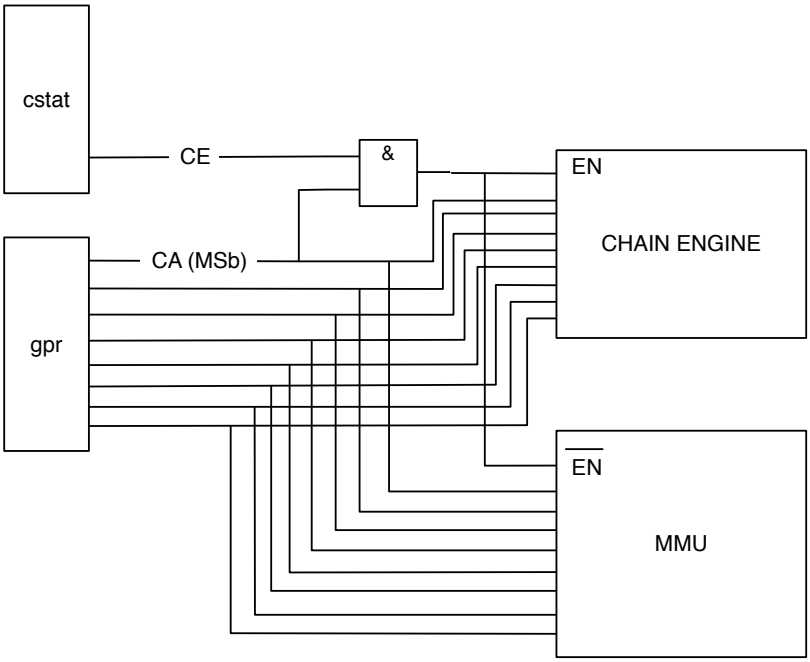


Figure 5.1: Logic for detecting chainable addresses using a dedicated bit. In order to allow the enabling of chaining support per application, a dedicated chain status register (cstat) has a special bit that allow the enabling / disabling of the chaining engine. If this bit is set and the chain address bit is set in the address located in one of the GPRs, the the address is routed to the chain engine instead of the MMU.

If the implementation supports 48-bit virtual addressing the lower half of the canonical range goes from 0 to $2^{47} - 1$ (or `0x00000000_00000000` - `0x00007fff_ffffffff`) while the upper half goes from $2^{64} - 2^{47}$ to $2^{64} - 1$ (or `0xffff8000_00000000` to `0xffffffff_ffffffff`). Although not detectable using a single bit, the in-between range is easily isolated using an expression like: `(p & 0xffff0000_00000000` $\neq$ `0xffff0000_00000000 && (p & 0x00008000_00000000)`.

Note that while it is possible to declare a range of non-canonical addresses as being chainable, if we want to maintain backward compatibility with future processors that may extend the number of bits used for virtual addresses, the

```
object_t *foo = 0x10000001; // The object foo is aligned at
                            // sizeof(int), set The LSb to
                            // indicate chainable
foo->field;                 // Load of 0x10000001 + 4

char bar[] = 0x20000001;    // The bar array is aligned at
                            // sizeof(char) == 1
bar[4];                     // Load of 0x20000001 + 4
```

Figure 5.2: Example illustrating the difficulties of dedicating the LSb for chainable pointer detection. Assuming foo->field is the same size as the elements of bar, there is no way to distinguish the loads from each other. Even though, pointer sized loads can be detected as they will be misaligned, data field loads of arbitrary sizes, should also be detected as belonging to a chain or not, and this is not possible.

chainable range must be readable by software. Such a readable range can be used by the allocation libraries that need to determine which addresses are available for the user program. In this case, it may also be possible to expose these registers as writable and simply let the software declare which range is chainable.

Note also that a valid virtual address must ultimately be mapped to an actual physical address, and that this also applies to chainable addresses. Under normal conditions, this mapping is done by page tables, and it is possible to also carry out the chainable address detection using an attributed page table.

Using Page Tables and TLBs

In this section we discuss the use of page tables to detect chainable memory regions. For the discussion, we use the terminology of the SRMMU[1] design, with the exception that instead of the term PDC (Page Descriptor Cache), we use the more familiar term TLB (Translation Lookaside Buffer). This does not mean that the MMU is assumed to be an SRMMU. We assume a certain amount of generality in the discussion and the solutions as discussed are equally applicable to most architectures around, including ARM, MIPS, PowerPC, SPARC, and x86. All of these architectures use multilevel page

[1]SPARC Reference Memory Management Unit

tables and most of them support *large pages*[2].

A memory management unit has two key ingredients, *page tables* and *TLBs*. Page table are divided in multiple levels where each subsequent level describes smaller regions in a hierarchy. The Level 1 page table (L1 PT) consist of entries known as *Page Table Pointers* (PTP) and *Page Table Entries* (PTE). The function of the PTPs are to point to next-level page table, that is in the L1 PT, the PTPs point to L2 PTs. A PTE, points to a specific memory page. Its virtual address is based on the combined multilevel index of the page table entry and its physical address is encoded in the PTE itself.

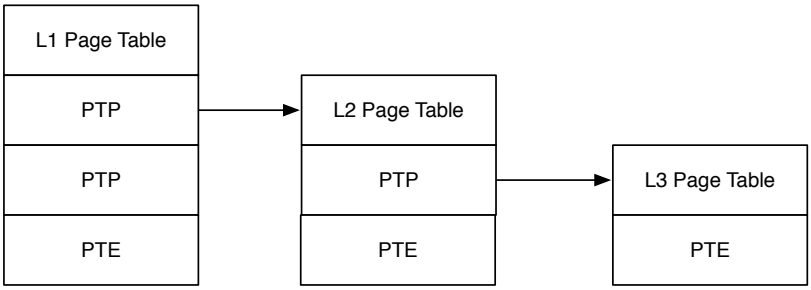


Figure 5.3: Multi Level Page Table

Depending on the level in which they are located, the PTEs describe different-sized pages. For example in the SRMMU, the describable regions are 4 GiB, 16 MiB, 256 KiB and 4 KiB [46]. The remaining bits in the virtual address (the ones not used for locating the PTE) are used in order to specify the offset within the page (this is possible as pages are aligned based on their size).

The PTDs and the PTEs assumes alignment of both page tables and pages. This is used to store various attributes such as read, write and execute permissions of the pages. The PTE attributes can be extended with a chainable bit. However, if this is only done on the PTEs, the TLB will in general make this impractical since it requires the TLB to have all of the PTEs for the given region. This follows from the fact that the TLB cache is of limited size (around one hundred entries). The advantage of a small TLB is that it provides a reasonably fast and energy efficient content addressable memory for looking up the most commonly used PTEs in sub-cycle time (in the case a PTE is not available in the TLB, the processor must walk the page table to find the

[2]Mappings where a virtual and physical page is larger than the smallest region

relevant PTE which causes a number of additional memory instructions to be issued).

The solution to the problem is simple. Instead of only placing the chainable attribute in the PTEs, the attribute can also be placed in the PTPs. We can therefore cache the chainable attribute bit for larger regions, without using as many entries in the TLB as would be necessary for caching the virtual to physical translations. In addition, when several chainable memory regions are supported, the identifying bits can be stored in page tables. These can then refer to a specific chainable memory table that track the object layout parameters.

Performance Implications

Neither of the systems have any major performance implications. In the bit dedication mechanism, identifying addresses as chainable is a very fast sub-cycle operation using very little logic. In the page table based system, there may be a performance implication, since for instant access, the objects must be located on a page whose address mapping is cached in the TLB. However, as the bit that identifies a page as containing chainable objects is stored on higher level page tables, the number of TLB entries can be minimized. A memory allocator must also be smart enough to align the actual mapped number of pages to the number that minimize the number of TLB entries needed to quickly lookup the chainable address bit.

Using the SRMMU as an example, a 2 GiB region is easily described using 128 TLB entries (or 1 TLB entry for a 4 GiB region). For the applications that we have explored, the data structure sizes are far smaller than this and it is sufficient to use only two entries of 16 MiB each. These two entries are likely to be located in the TLB.

In the experimental evaluation in Section 5.4, both methods are assumed to identify chainable addresses in sub-cycle time, so the performance penalty for either method is set to 0.

5.1.2 Tracking Active Pointers

As mentioned before, the chain engine resolves the chain addresses by embedding a forwarding pointer in each object. It is not practical to go through the chainable memory every time an object is accessed. Doing so defeats the purpose of the entire system. Consequently the lookup of a chain address must be optimized.

The chain engine speeds up the lookups by tracking active pointers on the stack (global object pointers are assumed to be the bottom of the stack). An active pointer is a pointer that is used in order to carry out loads within the chainable memory. To a certain extent, they have a role similar to a root pointer in a garbage collection system.

The active pointers are used to quickly execute a virtual-to-chain memory translation. It is important to note that this cached translation from a virtual address to a chain address is not associated with the virtual address directly, but is associated with the pointer that contains the address.

An active pointer record $A_{ptr} = \{V_{ptr}, C_{ptr}, C_{offset}, C_{len}, F_{offset}, F_{count}\}$ consists of the current virtual address (i.e. the pointer contents), a chain pointer, a chain offset, the chain length associated with the chain pointer, a field offset and a field access count. The chain pointer is the physical address in the chain memory of the relevant chain, the chain offset identifies the object within the chain and the chain length is the number of objects stored in that chain. Note, that this is not to be seen as a normal physical address that goes out to the normal memory bus, but it does describe the exact location of the chain in chain memory.

The C_* parameters are also stored in the chainable memory as special tags in the objects. This is necessary to resolve and access an object through its virtual address once the object is chained. It is up to the memory allocation system and compiler to add this field to each object. However, the hardware will manage it.

A complication is formed by the fact that these pointers are stored on the stack and in registers. In addition to this, the pointers are continuously copied back and forth between registers and the stack. The active pointer system has one record per general purpose register and one record per identified stack address. In the experiments described in Section 5.4 it is seen that the number of active pointers is very small in the benchmarks. Although, a recursive program may easily blow this out of proportion, we note that the support of deeply recursive programs is not a goal in this initial exploration. The flow of pointers between the stack and registers is illustrated in Figure 5.4.

By monitoring all loads through the stack pointer we can intercept the loads of pointers that may be referring to the chainable memory. When these pointers (now in registers) are in turn used for loads we can be sure they are being used to load values on the heap. These heap values do in turn need to be identified as pointers, one method that was suggested in [10] is based on comparing the most significant N bits of loaded value with the address of the loaded value. We do not use this approach, but actually check whether the full loaded value may be a pointer. In this case, we declare the register or the

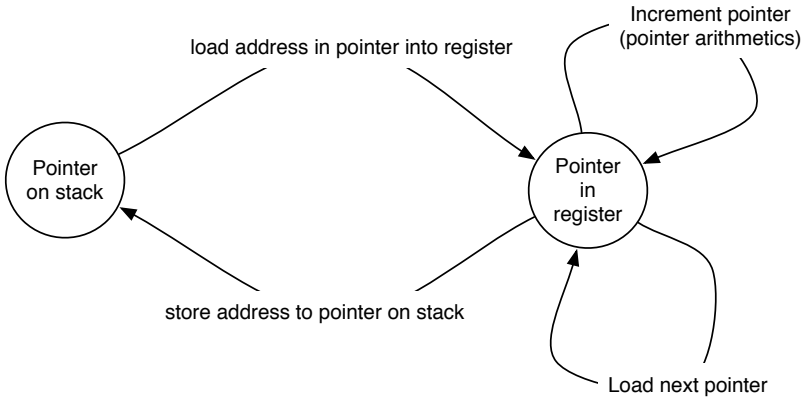


Figure 5.4: Flow of Pointers Between Stack and Registers

stack location where the pointer was stored as potentially active. It is finally declared active if it is used as the base address for another load instruction, which in turn guarantees that the potential pointer is a pointer[3]. The flow of the algorithm is illustrated in Figure 5.5.

Save and Restore

A problem with tracking active pointers is that pointers go out of scope whenever a function returns. In addition to this, if many functions are called in different calling contexts, many more active pointers are registered than is necessary.

This is solved by taking inspiration from the SPARC architecture. The SPARC architecture utilizes two instructions: *save* and *restore* to mark the entering and exiting of functions. Similar mechanisms can be used to automatically remove obsolete active pointers that are no longer in the programs scope. A *save* instruction will mark a new active pointer frame, and a *restore* instruction will kill all the active pointers that have been discovered since the last *save* instruction.

Note the *save* and *restore* can be embedded in, for example, the *call* and *return* instructions of the architecture.

[3]If it looks like a chainable pointer, it potentially is one. If it behaves as a chainable pointer it definitely is one.

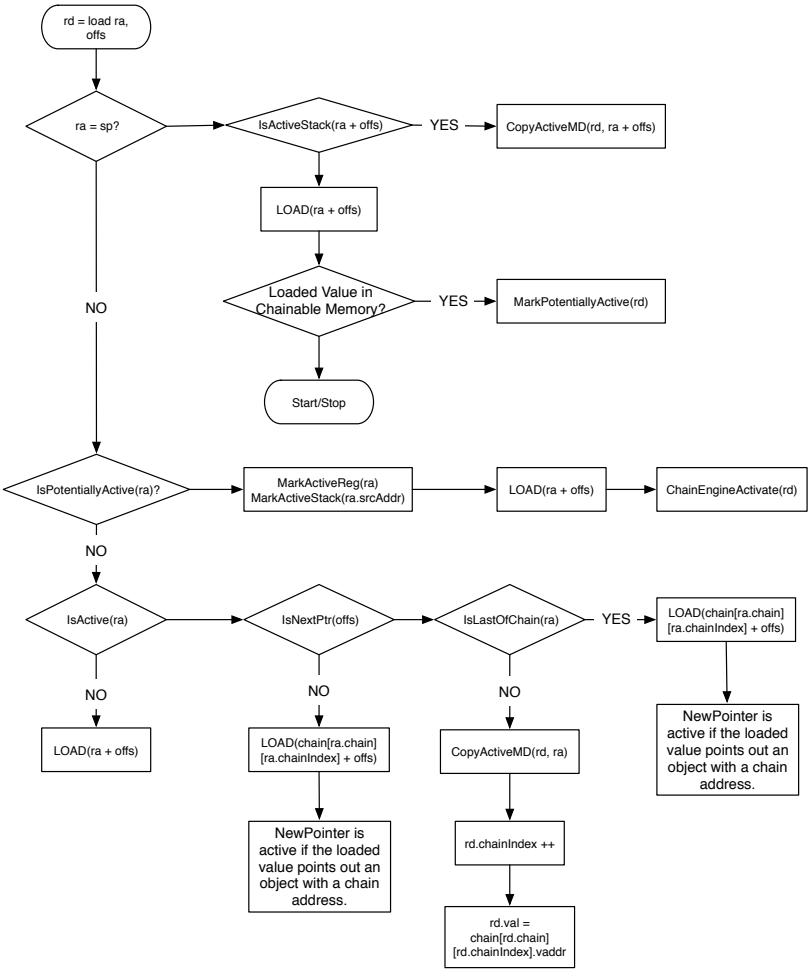


Figure 5.5: Active Pointer Tracking

5.1.3 Chaining

The chaining operation is triggered when an active pointer has been used for traversals in the same direction at least twice[4]. During the chaining operation, the objects in chainable memory are traversed using the pointer offset that has

[4]Experiments have also been carried out with immediate chaining

now been used for traversing the objects. All the visited objects are copied into chain memory creating a linearized copy of the original structure. For subsequent traversals of the same chain, the chained objects are accessed in the linearized order in chain memory, thereby increasing the spatial locality of the pointer chains.

In order to avoid traversing infinitely long chains, the chaining engine traverses at most a fixed number of objects in chainable memory (we have set this constant to 32) and it stops the traversal when it reaches an object that is already in chain memory (which is identified by the forwarding pointer) or a NULL pointer.

In order to perform the copying, the chain engine must know the size of the objects in the chains, this size is set using a system call by the application. In addition to the size, the chain engine also knows which fields in the objects are recursively typed pointers. This is stored in a bit vector. It is assumed in this case that all recursively typed pointers within the objects are aligned based on their size. This alignment restriction implies that if bit 1 is set in the bit vector, then the first field is a pointer. Using a 32-bit, bit vector, means that we can work with objects with sizes of up to 256 bytes, which is a reasonably large object.

During writes, the chaining engine evicts the tail of the chain if a chained pointer is overwritten. The eviction is done by traversing the chain starting at the current active pointer used for the write, copying all the objects after the current one back to the chainable memory.

5.2 Accessing Chained Objects

As already mentioned, the program works with normal virtual addresses. However, the active pointer system bypasses the normal memory system in most cases. There are four basic operations for accessing an object: load pointer, store pointer, load data field and store data field. For all of these, the following holds if the access is done through a non-active pointer. When accessing an address A, the processor inspects the address and determines if the address is in chainable memory. If it is, the first field of the object is loaded and checked as to whether it is a forwarding pointer or not (the first field being non-zero). If it is a forwarding pointer, the access is immediately forwarded to the chain memory. If it is not a forwarding pointer, the access is routed to normal memory. Note that at this time, the register with the base address of the object will likely be promoted to *potentially active*.

If an access is carried out through an active pointer, the following things

can happen. Firstly, if it is a load or store of a data field, the access is routed to the chain memory. Secondly, if we are accessing a pointer, the pointer may be a "next"-pointer, in this case the chain index is incremented in the destination registers active pointer record. If this index overflows the length of the chain, the next load brings in the object from main memory and accesses the forwarding field. If not the "next"-pointer, the value access is treated like a data access.

5.3 System Model

This section connects all the parts and gives a full system overview. In the chaining system all general purpose registers are extended with the chain location data as hidden registers. Figure 5.6 shows the additions of registers (visible and hidden ones) to the CPU. The special purpose registers are visible to user software. Of these, the frame ID identifies which active pointer frame is in use, this value is decremented on the issuing of the *restore* instruction and incremented on the issuing of a *save* instruction. The active frame in turn identifies the range of active pointers that are associated with the current frame. In the figure the current frame consist of active pointer entry number 0 to inclusive entry number 1. [5]

The active pointer file is a content addressable memory whose access key is the stack location of the corresponding pointer. The active pointer file contains the various fields that are associated with the active pointer. These include among others the *chain address* (CA), the *chain offset* and the *chain length*.

Each general purpose register is paired with its own active pointer status. Whenever an active (or a potentially active) pointer is loaded, the active pointer file entry is copied to the registers active pointer fields. The opposite holds for stores, in which case an active pointer record in the current frame may be allocated if an active pointer record for the stack location does not exist (the allocation is done by incrementing the current *top* register).

In addition to the CPU registers, the processor needs to store the chains somewhere. It would in this case be wise to divide the chain storage in multiple levels, in the same way normal caches work. The main difference is that instead of pulling in cache lines of smaller sizes, the chain cache would work using full chains (in our case of 32 objects). The chain cache would be indexed by the chain address.

[5] The *bottom* entry is equal to the *top* entry of the previous frame plus one, so the concrete realization does not have to implement both registers.

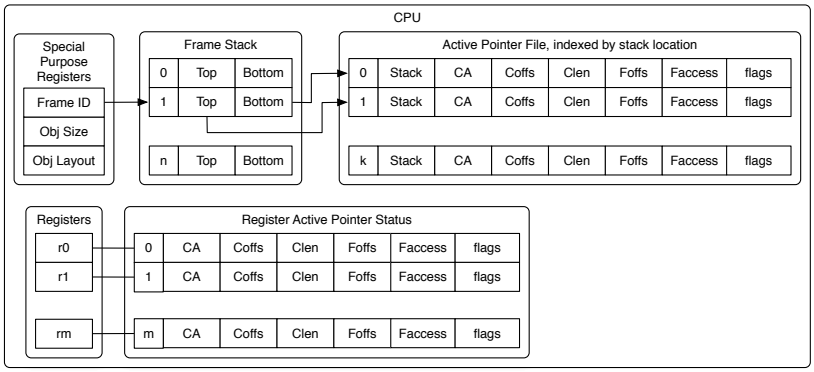


Figure 5.6: Additional registers in the processor used by the chaining system. The general purpose registers are called *rX* in the figure, these already exist in the processors, and only serve to illustrate how they are associated with the active pointer register status file. In the active pointer file, the stack field is the key and is used for lookup.

5.4 Experiments

This section empirically investigates the behavior of the chaining engine and the active pointer tracking. The experiments consist of instrumented applications that trace pointer accesses and a simulator running through the traces. The applications are taken from the SPARSE lib / SPARK00 benchmark suite and from the SPEC benchmark suite.

Three programs were instrumented: *Spmatvec*, a sparse-matrix-vector multiply code; *Dsolve*, a solver working with LU-factorized matrices; and *MCF*, a minimum constraint flow solver targeting the optimization of transports. The first two are from the SPARK00 suite of benchmarks and both work with the same data structure, an orthogonally linked list, where each node represents a separate cell in the matrix. The two codes traverse the matrix in different ways, *spmatvec* traverses the matrix row wise and *dsolve* traverses the upper and the lower half of the matrix in different directions. Neither *spmatvec* nor *dsolve* make any modifications to the matrix once it is constructed.

The *MCF* data structure is a tree, where siblings are linked in a binary linked list, and each node has a pointer to the left-most child, and to its parent. This tree structure undergoes optimization during the execution of MCF, resulting in many writes to the different pointer fields. *MCF* is known

Pointer Field Load	Source Pointer Destination Pointer Source Object Pointer Loaded Pointer Value Field
Value Field Load	Source Pointer Source Object Pointer Field
Pointer Field Store	Destination Object Pointer Field
Value Field Store	Destination Object Pointer Field
Pointer Copy	Source Pointer Destination Pointer Source Object Pointer
Load	Destination Pointer Loaded Pointer Value
Save	Function Pointer
Restore	Function Pointer

Table 5.1: Trace Formats

to be difficult to optimize and serves as a worst case scenario.

The purpose of the simulator is to explore the mechanisms and some properties related to the performance of the applications (such as hit-ratios of the chain accesses, and a simplified cost model) and not to measure actual performance.

5.4.1 Traces

The traces used by the simulator are based on a number of primitive operations used in the three applications. The operations include: pointer field loads and stores, value field loads and stores, pointer copies, loads of explicit values (i.e. not a load of a field) and the save and restore markers that indicate when a function is entered and exited.

The binary trace format for each operation was made identical in order to speedup the simulation (i.e. the same size and field divisions). The fields for each traced operation are however interpreted differently depending on the operation. Table 5.1 shows the relevant fields included in each trace entry.

Memory Type	Size	Latency (cycles)
L1 cache	32 KiB	4
L2 cache	256 KiB	11
L3 cache	8 MiB	30
RAM	N/A	162

Table 5.2: Performance Parameters. For the L3, the 8 MiB size is the default value. The size is varied in the experiments.

Test	Bandwidth
memcpy (read)	4.86 GB/s
memcpy (write)	4.86 GB/s
spmatvec (read)	219 MB/s
dsolve (read)	211 MB/s
mcf test (read)	276 MB/s
mcf test (write)	2 MB/s

Table 5.3: Memory bandwidth utilization tests on a Core i7 @ 2.7 GHz.

5.4.2 Performance Model

In order to estimate the execution time, a simplified performance model has been developed. The performance model is based on the use of caches and estimates for prefetching ability for the chain engine.

The processor model is based on an Intel Core i7. The default values selected are shown in Table 5.2. The cache latencies are taken from [16] while the RAM access time is taken from page 22 of [40] with the assumption of a 2.7 GHz processor. In addition to these values we have made a rough estimate of memory bus bandwidth by timing the memcpy function for a best case value, and by analyzing the actual amount of data traffic (recorded in the traces) from the benchmarks detailed below. These tests yielded the parameters listed in Table 5.3.

From the data in Table 5.3, we conclude that the memory bandwidth is substantially underutilized on the relevant tests.

Chain access time is set to be equivalent to L1 accesses, as they are assumed to enable substantial opportunities to prefetch chains into memory closer to the processor. Note that this prefetching mechanism is not fully modeled in the simulator at present.

Note that each memory load for a non-active pointer (when the address is

in chainable memory) results in two memory loads: firstly of the forwarding pointer embedded in the object and secondly of the actual value, unless the forwarding pointer was valid, in which case the value is taken from the chain memory. Since it is assumed that the cache and chain memory have the same speed, the initial load of the forwarding pointer will bring in the larger part of the object into cache, thus the actual field load will have the same speed, independent of whether it goes to chain memory or stays in normal memory.

5.4.3 Simulator

The chaining simulator is responsible for reading the traces and executing the traces as pseudo-instructions. The load and store instructions in the traces are broken up into a number of micro operations. A pointer field load for example, turns into a stack load of the pointer (into a register) followed by a load with offset using the register as base pointer, and a stack store where the loaded pointer is stored to the relevant variable on the stack. This approach, emulates the behavior of program running on a normal load store architecture[6].

On stack loads, the simulator looks in the active pointer file and checks if the stack location is registered. If it is not, the load is carried out as normal. If it is, the active pointer record is copied to the register's own active pointer record.

Consequently on stack stores, the register's active pointer record is moved to the active pointer file and associated with the relevant address. This association happens if the stored value is *potentially active* or *active*, but not if the stored register is *clean*.

While running through the traces, the simulator not only builds up the active pointer file and the records for the registers, but also constructs any chains and records a number of statistics regarding the different events. For example, the number of stack loads of active pointers, the number of chain hits and misses.

A requirement for running the simulator, is that a memory image has been constructed before this and maintained during the simulation (i.e. updated when the chains are modified), the memory image is necessary for traversing and building the chains. This memory image is constructed for a given program trace by running through the trace, and recording the first loaded value from every address. This memory map is then loaded when the actual simulation is executed.

[6]We count x86 as a load store architecture as this is the internal behavior, even though it is not visible to the programmer.

5.4.4 Results

Results are presented in Tables 5.4, 5.5, 5.6 and 5.7. The first three tables detail the statistics for the chaining engine. Table 5.7 shows the actual speedup when applying the performance model discussed in Section 5.4.2.

The numbers in Tables 5.4, 5.5 and 5.6 should be interpreted as follows. The total number of chained objects, is the total count of objects that have been placed in a chain. This number may be larger than the total number of objects in the data structure if evictions take place. If the data structure is not modified in a way that will trigger an eviction event, the number of chained objects will always be less then or equal to the total object count.

Object layout is the bitfield describing which fields are pointers or not. The data field loads is the number of loads of data values in an object. The number of loads in a chained object is also listed with the fraction of such loads compared to the total. For pointer field loads, the total is all pointer field loads, the "not next" loads are loads on a pointer that is not considered the "next" pointer for the object. "Next (EOC)" describes the number of loads of the "next pointer" that was carried out at the end of the chain.

Pointer field stores to the "next" pointer will potentially result in a chain eviction, if the written pointer field is not the last object in the chain.

In Table 5.7 the estimated cycles needed for memory operations, based on the performance model in Section 5.4.2 is listed together with the speedup of applying the chaining (and caches) compared to a running a normal code with only the cache.

Note that the data in Table 5.7 excludes the time spent on doing calculations, since this is the same independently of whether or not the chaining engine is in use.

5.5 Discussion

As can be seen in Table 5.7, substantial speedups in line with prior research in the area was achieved using the simulated hardware model. *SPMATVEC* and *DSOLVE* performed especially well.

For *MCF-test*, the slowdown is considerable. However, this slowdown is caused by the *test* data set fitting in L1 cache, essentially eliminating all possible advantages of the chaining mechanism.

This effect of cache is also seen for *SPMATVEC* and *DSOLVE*. As the amount of L3 cache goes up, the actual speedup goes down, since more of the problem now fits in the cache.

Object	Total number of objects	1 294
	Object size	120
	Object layout	00111100_2
Chain	Total chained objects	16 290
	Number of chains built	6 765
	Average chain length	2.4
	Eviction Events	6 734
	Evicted Objects	15 146
Data Field Loads	Total	3 114 184
	In chained object	2 523 438 (81.0 %)
Pointer Field Loads	Total	1 984 057
	Not Next	957 878 (48.3 %)
	Next	293 684 (14.8 %)
	Next (EOC)	432 002 (21.8 %)
Pointer Field Stores	Next	8 058
	Not Next	18 088

Table 5.4: Chaining Simulator Results (MCF)

Object	Total number of objects	1 143 140
	Object size	32
	Object layout	1010_2
Chain	Total chained objects	1 120 088
	Number of chains built	45 298
	Average chain length	24.7
	Eviction Events	0
	Evicted Objects	0
Data Field Loads	Total	22 862 800
	In chained object	22 355 656 (97.8 %)
Pointer Field Loads	Total	11 431 400
	Not Next	0 (0 %)
	Next	10 724 848 (93.8 %)
	Next (EOC)	452 980 (4.0 %)
Pointer Field Stores	Next	0
	Not Next	0

Table 5.5: Chaining Simulator Results (SPMATVEC)

Object	Total number of objects	1 143 140
	Object Size	32
	Object Layout	1010_2
Chain	Total chained objects	717 356
	Number of chains built	105 967
	Average chain length	6.8
	Eviction Events	0
	Evicted Objects	0
Data Field Loads	Total	24 019 340
	In chained object	23 690 937 (98.6 %)
Pointer Field Loads	Total	12 355 450
	Not Next	1 878 353 (15.2 %)
	Next	8 610 984 (69.7 %)
	Next (EOC)	1 566 300 (12.7 %)
Pointer Field Stores	Next	0
	Not Next	0

Table 5.6: Chaining Simulator Results (DSOLVE)

Bench	Cache Only	Chaining	Build	Speedup
MCF test	29159320	32375280	1059759 / 1001551	0.90
MCF train	8981414378	6845304321	17480519 / 18349806	1.31
SPMATVEC	1762582495	328398003	181149328 / 0	5.37
DSOLVE	1663239273	259639399	116652311 / 0	6.41

Table 5.7: Memory Access Times (In Cycles), for default test. Caches: 32 KiB L1, 256 KiB L2, 8 MiB L3. Chaining after 2 accesses along active pointer.

Bench	Cache Only	Chaining	Build / Evict	Speedup
MCF test	29159320	33861792	924132 / 857997	0.86
MCF train	8981414378	6994648303	15259068 / 16015148	1.28
SPMATVEC	1762582495	330699648	181518457 / 0	5.33
DSOLVE	1663239273	254536680	113664778 / 0	6.53

Table 5.8: Memory Access Times (In Cycles), for default test. Caches: 32 KiB L1, 256 KiB, 8 MiB L3. Chaining after 1 access along active pointer.

a	Cache Only	29159320
b	Chain Total	32375280
c	Chain Build	1059759
d	Chain Evict	1001551
e	Total - Build - Evict (b-c-d)	30 313 970
f	Difference (e-a)	1 154 650

Table 5.9: Components of MCF test memory access time.

The average chain length of *DSOLVE* is less then half of the average length of *SPMATVEC*. This may seem surprising when considering that the two codes were executed with the same matrix as input, but it is natural when one considers that to construct a chain, the active pointer's field access counter (F_{count}) must be saturated. For this reason the first few elements on the rows (starting with the diagonal elements) and the first few elements on the columns (also starting with the diagonals), are not chained. For *SPMATVEC* in contrast, only a few elements per row will remain un-chained. From this it is possible to conclude that when reducing the time it takes to start building chains (to do it immediately after the first access), there will likely be a negligible or small positive impact on *SPMATVEC*, and a larger positive impact on *DSOLVE*, since it will bring in relatively more elements into the chains than a change in *SPMATVEC* will. For *MCF*, as the access-patterns do not follow a straight chain, we should expect a negative impact, depending on the input data structure. By looking at the numbers in Tables 5.7 and 5.8. We can conclude that these assumptions are valid, although a slight degradation in the relative performance for *SPMATVEC* can be observed.

The primary reason for this is that the chains are limited to a fixed number of objects. This means that for a chain that is actually over this limit, additional chain segments are built which results in a performance penalty. Adding one additional object increases the proportion of chains that overflows. For SPMATVEC this proportion in the test case used, changed from 452980 (0.039626) to 458080 (0.040072)[7].

It should be noted, that the simulator takes into account prefetching abilities of the chains, which is used for the chain access time estimate. In fact for the work done in [56], the processor's prefetching engine was a significant source of the improvements in performance after the restructuring operations. Since the restructuring made the memory reference streams predictable, the

[7]The second number not listed in Table 5.5. It is given here in order to allow for the comparison

Bench	L1	L2	L3	No Chain	Chaining	Speedup
SPMATVEC	32 KiB	256 KiB	8 MiB	1762582495	328398003	5.37
DSOLVE	32 KiB	256 KiB	8 MiB	1663239273	259639399	6.41
MCF test	32 KiB	256 KiB	8 MiB	29159320	32375280	0.90
MCF train	32 KiB	256 KiB	8 MiB	8981414378	6845304321	1.31
SPMATVEC	32 KiB	256 KiB	32 MiB	679704209	271451537	2.50
DSOLVE	32 KiB	256 KiB	32 MiB	525401999	240916274	2.18
MCF test	32 KiB	256 KiB	32 MiB	29158480	32374412	0.90
SPMATVEC	32 KiB	256 KiB	64 MiB	511826703	268833783	1.90
DSOLVE	32 KiB	256 KiB	64 MiB	525402096	240916416	2.18
MCF test	32 KiB	256 KiB	64 MiB	29152586	32369939	0.90
SPMATVEC	64 KiB	256 KiB	8 MiB	1762714248	328317323	5.37
DSOLVE	64 KiB	256 KiB	8 MiB	1662932678	259587480	6.41
MCF test	64 KiB	256 KiB	8 MiB	26798948	31083885	0.86
SPMATVEC	32 KiB	256 KiB	1/33 MiB	621978165	358096428	1.73
DSOLVE	32 KiB	256 KiB	1/33 MiB	525406648	269187869	1.95
MCF test	32 KiB	256 KiB	1/33 MiB	29159005	32371815	0.90

Table 5.10: Speedup based on Cache Sizes. For the last three tests, the chaining version received 1 MiB cache, while the normal version received 33 MiB cache.

prefetching mechanisms could start detecting the memory reference streams and improve performance.

Prefetching of normal memory is not simulated by the simulator described in this chapter. However it is not likely that a full hardware implementation will have higher performance because of this without taking into account the pointer chain prefetching mechanisms explored by others.

The mechanism can potentially be improved by taking longer histories into account. For example, a chain should perhaps not be constructed based on a single direction only, but instead based on history information, similar to the ways branch prediction is handled.

Chapter 6

Conclusions

This thesis explored different approaches to data restructuring and showed that that the software and hardware approaches resulted in substantial speedups. This leads to a number of conclusions:

Firstly, compiler and runtime assisted automatic data restructuring is practical for well behaved and reasonably simple codes. Even when carrying out expensive runtime tracing, it was possible to get a speedup for iterative codes.

Secondly, we can conclude from the NP-completeness of the Minimum Confined Components problem, that in order to carry out restructuring on grids, it is not enough to simply analyze the pointer linked nodes at runtime. Additional information is needed, this can be provided by the compiler (through analysis of code) or the programmer (by providing more information to the compiler or the runtime). In addition to this, more advanced restructuring could be enabled. For instance if it is known by the compiler that a pointer linked data structure represents a grid, the compiler could apply loop interchange on some pointer linked codes, or it could reorder the data according to what is best for the inner loop.

Thirdly, for more complex restructuring tasks, where applications have multiple access patterns and a compiler cannot automatically choose which one, language extensions can be used that assign rules for how data types may be used. Rules like this are often implicit in todays code's, and explicit encoding allows the compiler to find additional issues with the code that had otherwise gone unnoticed.

Fourthly, for highly dynamic data structure, active restructuring is possible provided sufficient hardware support is given.

There is a lot more work to be done in this area. Pointer based grids should

for example be supported by compilers directly. The extensions described in Chapter 4 do at present not support the notion of orthogonality as introduced in Chapter 3. This could be remedied, but it is not entirely certain how additional orthogonality attributes can be verified in a way that would keep the performance penalty at an acceptable level. We do however know, from Chapter 3 that an assumption of orthogonality can be verified in polynomial time, the question here is whether the given algorithm runs fast enough and for which programs it is fast enough for, it may also be possible for the compiler to statically detect detect whether the assumption holds in a conservative way. In addition to this, transformations and optimizations based on this notion should be developed.

With respect to the results described in Chapter 5, the chain memory needs further evaluation and development. Firstly a more high fidelity simulator is needed in order to increase the accuracy of the simulation results and in order to run actual code, and not just address access traces. After this step has been carried out, an actual implementation in register transfer notation could be explored and integrated with an existing processor.

Bibliography

[1] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In *PLDI*, pages 269–279, 1998.

[2] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In Rajiv Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2010.

[3] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *PODS*, pages 20–29, 2006.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, pages 207–216, 1995.

[6] Kostas Bousias, Liang Guang, Chris R. Jesshope, and Mike Lankamp. Implementation and evaluation of a microthread architecture. *Journal of Systems Architecture - Embedded Systems Design*, 55(3):149–161, 2009.

[7] Fred Buckley and Frank Harary. *Distance in Graphs*. Perseus Books (Sd), January 1990.

[8] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS*, pages 40–52, 1991.

[9] Jamison D. Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *MICRO*, pages 62–73, 2002.

[10] Robert Cooksey, Stéphan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, pages 279–290, 2002.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Mass., USA, 2001.

[12] Stephen Curial, Peng Zhao, José Nelson Amaral, Yaoqing Gao, Shimin Cui, Raúl Silvera, and Roch Archambault. Mpads: memory-pooling-assisted data splitting. In Richard Jones and Stephen M. Blackburn, editors, *ISMM*, pages 101–110. ACM, 2008.

[13] Haakon Dybdahl, Per Stenström, and Lasse Natvig. An lru-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. *SIGARCH Computer Architecture News*, 35(4):45–52, 2007.

[14] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. In *IPDPS Workshops*, pages 505–511, 2000.

[15] 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, Jun 2008.

[16] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, February 2012.

[17] Pascal Fradet and Daniel Le Métayer. Shape types. In *POPL*, pages 27–39, 1997.

[18] Hassan Ghasemzadeh, Sepideh Sepideh Mazrouee, and Mohammad Reza Kakoee. Modified pseudo lru replacement algorithm. In *ECBS*, pages 368–376. IEEE Computer Society, 2006.

[19] 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*, pages 1–15, 1996.

[20] Olga Golovanevsky and Ayal Zaks. Struct-reorg: current status and future perspectives. In *Proceedings of the GCC Developers' Summit*, pages 47–56, Jul 2007.

[21] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design*. Number 0-471-38365-1. Wiley, 2002.

[22] Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 69–92, 2005.

[23] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *MSP/ISMM*, pages 256–263, 2002.

[24] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. In *PLDI*, pages 249–260, 1992.

[25] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[26] Joseph Hummel, Alexandru Nicolau, and Laurie J. Hendren. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, pages 208–216. IEEE Computer Society, 1994.

[27] Yuan-Shin Hwang and Joel H. Saltz. Identifying def/use information of statements that construct and traverse dynamic recursive data structures. In *LCPC*, pages 131–145, 1997.

[28] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. http://download.intel.com/design/processor/manuals/253665.pdf, March 2008.

[29] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[30] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *POPL*, pages 196–205, 1993.

[31] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *IEEE PACT*, pages 268–279, 2001.

[32] Kenneth S. Kundert. Sparse lib. `http://sparse.sourceforge.net/`.

[33] Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli. Sparse lib. `http://www.netlib.org/sparse`.

[34] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization.* PhD thesis, University of Illinois at Urbana-Champaign, May 2005. `http://llvm.cs.uiuc.edu`.

[35] Chris Lattner and Vikram S. Adve. Automatic pool allocation for disjoint data structures. In *MSP/ISMM*, volume 38 (2 Supplement) of *SIGPLAN Notices*, pages 13–24. ACM, 2002.

[36] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, 2005.

[37] Chris Lattner and Vikram S. Adve. Transparent pointer compression for linked data structures. In *Memory System Performance*, pages 24–35, 2005.

[38] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

[39] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR*, pages 398–416, 1993.

[40] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel XeonTM 5500 processors. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.

[41] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.

[42] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD Conference*, pages 297–306, 1993.

[43] Shai Rubin, David Bernstein, and Michael Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *CC*, pages 259–273, 1999.

[44] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[45] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Computers*, 40(5):603–612, 1991.

[46] SPARC International, Inc. *The SPARC Architecture Manual*, 8 edition, June 1999.

[47] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[48] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[49] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[50] Harmen L. A. van der Spek. *Transparent Restructuring of Pointer-Linked Data Structures*. PhD thesis, University of Leiden, 2010.

[51] Harmen L. A. van der Spek, Erwin M. Bakker, and Harry A. G. Wijshoff. SPARK00. `http://www.liacs.nl/~hvdspek/SPARK00/`, 2007.

[52] 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.

[53] 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.

[54] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. Automatic restructuring of linked data structures. In *LCPC*, pages 263–277, 2009.

[55] Harmen L. A. van der Spek and Harry A. G. Wijshoff. SPARK00: A benchmark package for the compiler evaluation of irregular/sparse codes. Technical Report LIACS 2007-06, Leiden Institute of Advanced Computer Science, 2007.

[56] Harmen L.A. van der Spek, C.W. Mattias Holm, and Harry A.G. Wijshoff. A compilation framework for the automatic restructuring of pointer-linked data structures. *High-Performance Scientific Computing: Algorithms and Applications*, pages 97–122, 2012.

[57] Thomas Wolle and Hans L. Bodlaender. A note on edge contraction. Technical Report UU-CS-2004-028, Institute of Information and Computing Sciences, Utrecht University, 2004.

[58] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

Chapter 7

Samenvatting

In hoofdstuk 2 beschrijven we een nieuwe, algemene herstructurerings-aanpak voor de optimalisatie van data layout van pointer-gekoppelde datastructuren met behulp van compiler en runtime analyse en datastructuur herschrijving. Onze experimenten tonen aan dat met de herstructurering van pointer-gekoppelde datastructuren de prestaties aanzienlijk kunnen verbeteren, ondanks de overhead van het analyseren en herschrijven.

De theoretische fundamenten voor een gewoon pointer-gekoppeld datastructuur, de *orthogonally linked list*, of *sparse grid* wordt onderzocht in hoofdstuk 3. Grids zijn gewone datastructuren die in veel codes voorkomen, zoals bijvoorbeeld sparse matrix codes. Dit hoofdstuk introduceert nieuwe graaftheoretische concepten van *confined components* en *strictly ordered orthogonality*. Het Minimum Confined Components probleem wordt bewezen als zijnde NP-compleet waardoor het opsporen van arbitraire grids onpraktisch is zonder extra *a priori* opgelegde beperkingen. Hoewel de complexiteit in het algemene geval niet triviaal is, introduceren we een efficiënt algoritme dat *confined components* kan detecteren voor diverse soorten graafen.

In hoofdstuk 4 introduceren we Pax C, een volledig *backwards* compatible extensie van de C programmeertaal. De taalextensies maken de programmeur mogelijk om traversal patronen in de code uit te drukken op een natuurlijke manier, waardoor we een herstructurering kunnen doen zonder ingewikkeld analyses. De extensies zijn geëvalueerd op een aantal voorbeeld codes, sparse matrix codes en het Minimum Cost Flow programma (MCF) van de SPEC benchmark-serie.

Hoofdstuk 5 introduceert een dataherstructureringsmethode implementeerbaar in hardware. De methode is gebaseerd op een speciaal geheugengebied

dat door de CPU gebruikt wordt om pointer-gekoppelde objecten naast elkaar op te slaan. In dit hoofdstuk onderzoeken we het systeem en gebruiken we een simulatie-gebaseerde aanpak om de overhead van het herstructureringssysteem te vergelijken met de behaalde prestatiewinst. De resultaten laten zien dat de werkwijze correct is en minstens zo goed presteert als bestaande software-gebaseerde methodes. Een belangrijke bijdrage van deze methode is dat, in tegenstelling tot bestaande software benaderingen, de hardwaregebaseerde aanpak dynamisch bijgewerkte pointerstructuren op een automatische manier kan verwerken en dat deze methode werkt zonder wezenlijke wijzigingen van de programma's.

Chapter 8

Curriculum Vitae

Carl Wilhelm Mattias Holm was born on 3 June, 1980, in Täby, Sweden. As a self proclaimed computer geek, he started his B.Sc. in computer engineering at the University-College of Borås in 1999 and after also having spent a year serving in the 18'th armored regiment of the Swedish Army as an NCO during the year of 2000, he finally graduated in 2003, *first in class*. Following the B.Sc. degree, Mattias started his M.Sc. studies at Chalmers University of Technology which involved an exchange year at the Technical University of Delft in the Netherlands. He graduated during spring 2006 after having completed his master's thesis: *SPARCv8 Simulation with Simics*. He first kept on working for a while at Virtutech AB where he extended and optimized his master's thesis project. Following a few months of work at Virtutech, Mattias left for the Netherlands to take up a trainee position at the European Space Agency. He worked there for one year until taking up a job at SciSys in Bristol, UK. At SciSys, Mattias worked with the flight software for the SWARM satellite constellation. In October 2008, Mattias left SciSys and once again returned to the Netherlands, this time in order to start his Ph.D. at the Leiden Institute of Advanced Computer Science. At *LIACS*, Mattias has been carrying out research in the area of data structure optimization techniques and teaching in *Operating Systems* and *Networks*. In order to improve the operating systems course, Mattias, part of a team of two, implemented an operating system kernel from scratch. This kernel is now used and modified by students during course work.