The Design and Implementation of Compression Techniques for Profile Guided Compilation

by

Youtao Zhang

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 0 2

Get the official approval page
from the Graduate College
before your final defense.

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: _____

To my parents, Yurong Zhang and Xuexia Yuan.
To my wife, Jun Yang.

Acknowledgements

My special thanks are for my advisor Dr. Rajiv Gupta. I thank him for his guidance in selecting research topics and refining approaches. During the years in pursuing my degree, he always encouraged me and took great patience to help me in writing papers and giving presentations. It is a privilege to have him as my advisor.

I feel grateful for my wife, Jun Yang. She gave me huge support in these years while she had to work hard on her Ph.D. dissertation research. I wish her a successful career.

I would also like to take this chance to thank other members in my committee Dr. Peter Downey and Dr. Samuya Debray. I also thank Xiangyu Zhang and all the people who helped me over the years.

Table of Contents

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

List of Figures

List of Figures—*Continued*

List of Tables

Abstract

Advances in program profiling techniques have led to advances in compiler optimization techniques, and *vice versa*. This dissertation makes contributions in the areas of program profiling as well as profile guided optimizations. More specifically, it designs and evaluates a new compressed representation for profile data such that profile guided optimizations can benefit from it. A type-based value profiling technique is also developed such that new data compression techniques can be designed to exploit value redundancy present in program data.

A timestamped whole program path (TWPP+) representation is proposed to compress program traces which contain both control flow and memory address information. Instead of considering a trace as a stream of symbols, TWPP+ divides a complete trace into a control flow trace part and a memory dependence trace part; each part is then reorganized to allow fast retrieval of information during data flow analyses. Execution profiles can thus be integrated to help a broad range of compiler analyses and optimizations. Three different applications are shown to demonstrate the strength of this new representation.

A type-based value profiling framework is developed to help identify redundancy in data values and thus design new data compression techniques for improving memory behavior. Two types of redundancies are identified in representations of small values and pointer addresses respectively. Both software and hardware approaches are proposed and evaluated to exploit these opportunities. The software approach through data compression transformations greatly reduces the memory footprint and speeds up the program executions with the help of six specially designed data compression instructions. The hardware approach employs compression to enable partial cache line prefetching resulting in consistent improvements in the program's execution time and reduction in memory traffic.

Chapter 1
Introduction

Traditionally compile-time optimization algorithms are applied only in situations where it is known that the optimization is definitely applicable and will generate beneficial results. However, such a conservative approach fails to exploit many valuable optimization opportunities. A profile-guided optimizer uses the information of a program's past executions in two ways to aggressively optimize the program. First the profiles can be used to identify new optimization opportunities that are frequently observed during program execution but are not detected by static analyses. Second the profiles can be used to carry out sophisticated cost-benefit analysis to apply transformations that improve the performance of one part of the program at the expense of a performance loss in another part of the program.

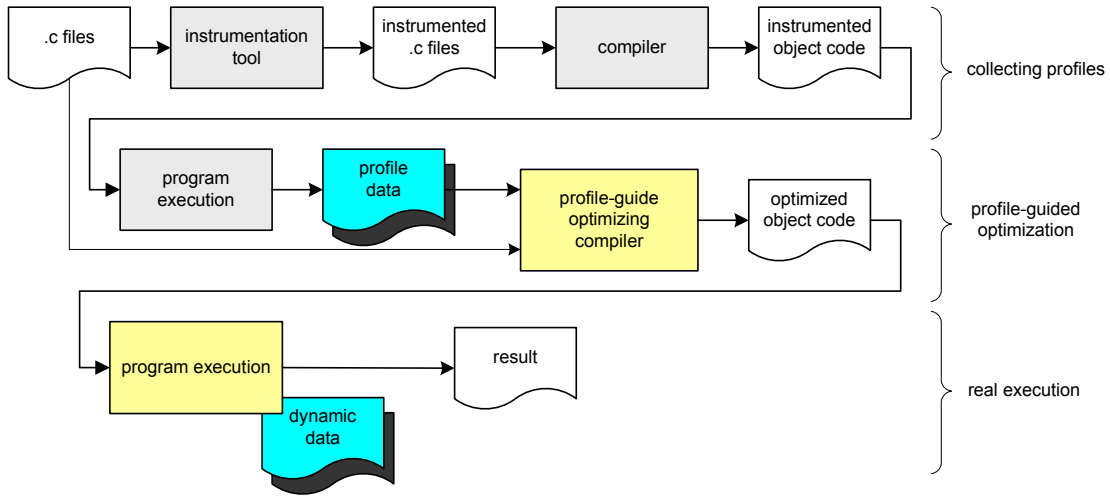


Figure 1.1. Overview of profile-guided compilation.

Figure 1.1 summarizes the steps of a typical profile-guided compilation. Before performing any optimization, an instrumented version of the program is generated.

The instrumented program is executed on one or more sets of representative inputs and the profiles for these executions are collected. With the help of profiles, an optimizing compiler recompiles the program and generates the optimized object code. The optimized object code is then used in all future executions with real inputs. Typically, *simple* representative sets of inputs are used in collecting profiles and profiling executions are much *shorter* than real executions. During the execution, the amount of profile data that is generated from a profiling execution is significantly *less* than that from a real execution.

1.1 Program profiling and profile guided optimizations

There is a close interaction between the research in program profiling and the research in the development of new profile guided optimization techniques. Advances in one area help create advances in the other, and *vice versa*.

On the one hand, one research trend in profiling is to collect more kinds of detailed and accurate profiling information from which more optimization opportunities can be discovered. Powerful optimization algorithms can then be developed to exploit these opportunities. On the other hand, with the rapid advances in computer architecture and system designs, many kinds of optimization opportunities are known to exist in many programs. However, profiling techniques are needed to collect information that can guide the design of cost-benefit analyses to effectively exploit these opportunities.

Let us consider the situations where profiling research has greatly influenced optimization research. Simple profiles were collected in earlier days and they worked well in finding more optimization opportunities than static analyses. For example node profiles consisting of execution frequencies of basic blocks in a control flow graph were collected. Compilers could be directed to optimize most frequently executed regions so that for a given fixed amount of compilation time, the improvement in program performance could be maximized. Slightly more complicated edge profiles,

which count the execution frequencies of each edge in a control flow graph, can be used to enable more complex optimizations (e.g., Young *et al.* [62] proposed the use of edge profiles for interprocedural branch alignment). More complex path profiles [4], which consist of execution frequencies of acyclic sequences of basic blocks are also collected. Gupta *et al.* [22, 24, 23] used path profiles to enhance traditional optimization techniques as well as develop new ones.

Now let us consider some situations in which optimization research has driven research into new profiling techniques. Programs and architectures are increasing in complexity and creating new challenges for developing optimizing compilers. Dynamically allocated data objects are frequently used and they often lead to poor cache performance. A better data layout scheme could greatly reduce the number of cache misses and improve the overall performance. However, to assist the design of different memory layout optimizations, new types of profiling techniques are needed. Calder *et al.* [13] suggested to collect a temporal relation graph (TRG) which summarizes the usage relationship between different objects. New memory allocation policy can then be designed to allocate affiliated objects close to each other. Recent research reveals that dynamic optimizations, which optimize the program during the execution, have many advantages. However, given the restricted runtime constraints, there is demand for new profiling techniques which are cheap and yet sufficiently accurate. To support optimization in a dynamic optimization environment, Arnold [2] proposed a counter-based sampling technique that can perform effective runtime profiling.

1.2 Overview of the research

This dissertation further illustrates the close interaction between research in profiling techniques and profile-guided optimization opportunities. It designs and evaluates compressed representation for profiling data allowing profile-guided optimizations to benefit from this advance in program profiling. The newly developed representation

is demonstrated to help in the design of new optimization algorithms. A type-based value profiling technique is also developed such that new data compression techniques can be designed to exploit value redundancy present in program data.

1.2.1 Representation of profiling data

Traditional compiler optimizations perform data flow analyses based on program control flow graphs. A recent advance in profiling proposed collection of the *whole program path* (WPP) profiles [32] which is a compressed form of the program's control flow trace. Although WPP contains complete and accurate dynamic control flow information, it can be up to several gigabytes in uncompressed form and hundreds of megabytes in compressed form. Information retrieval is very slow using WPP. As a result, it is difficult for compiler optimizations to take advantage of this new advance in program profiling. Moreover, data dependence information is needed for inferring certain data flow facts. In this dissertation, a new representation TWPP+ is proposed to address these problems. Given a complete program trace that contains control flow trace and address trace, TWPP+ explicitly separates the control flow and memory dependence information from each other. Each type of information is organized in a way that assists later compiler analyses and optimizations. Figure 1.2 compares this new representation with the whole program path (WPP) technique. While the WPP representation tries to achieve the highest possible compression ratio, the new representation puts more emphasis on accessibility, that is, the ease use of the information. Besides, the WPP representation does not consider dynamic memory dependence information which is also very important for some analyses and optimizations.

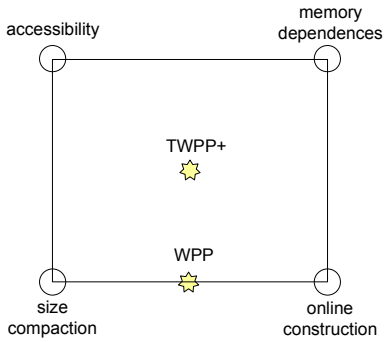


FIGURE 1.2. Comparison with whole program path.

1.2.2 Profiling for value redundancy detection

Over the last decade, while the processor speed has been improved 55% each year, the memory speed has been improved only 7% each year [43]. As a result, the memory system has become a major bottleneck in improving system performance. The situation is worsened by the fact that machine word size has increased from 8 bits to 64 bits. Recent research [64, 61] has found that there is a significant level of redundancy in dynamic value representation. Figure 1.3 shows that for a 32-bit machine, and for a spectrum of benchmark programs, on an average 59% of all accessed 32-bit values can be effectively represented by half of their original size, that is, using 16 bits.

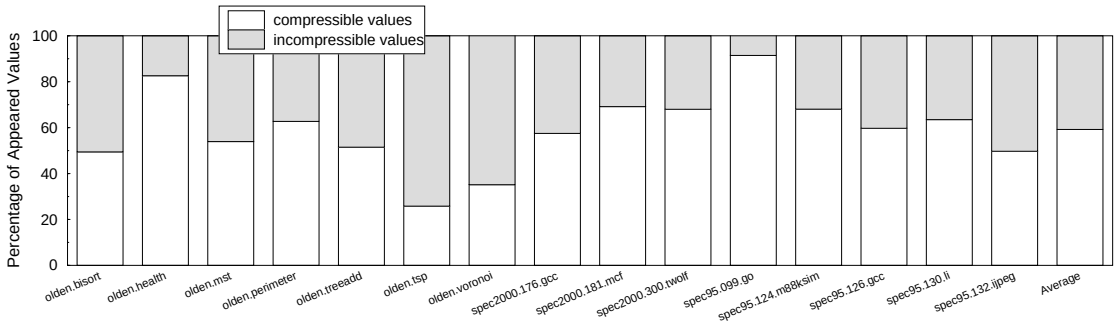


FIGURE 1.3. Values compressible to half of their size.

By removing redundancy through dynamic data compression, cache performance could be greatly improved. However, there are no available profiling techniques aimed at catching this type of optimization opportunity. Moreover, critical runtime con-

straints restrict the runtime application of traditional compression techniques. In this dissertation, a new type-based profiling technique is developed to assist in the design of new dynamic data compression techniques. The potential of the new compression is further exploited through both software and hardware techniques.

1.3 Organization

The rest of this dissertation is organized as follows. Background research on program profiles and profile-guided optimizations is presented in Chapter 2. A timestamped whole program path (TWPP+) representation for compressing program profiles is developed and evaluated in Chapters 3, 4 and 5. Chapter 3 discusses how to compress control flow traces into TWPP. Chapter 4 enhances TWPP to include compressed memory dependence profiles. Three applications are discussed in Chapter 5 to illustrate the use of information contained in a TWPP+ representation.

A type-based profiling technique for finding value representation redundancy is proposed in Chapter 6. Using the data collected from profiling, both software and hardware data compression techniques are developed to exploit the opportunities in removing value representation redundancy. The software approach based upon data compression transformations, is discussed in Chapter 7. The hardware approach that employs compression to enable partial cache line prefetching, is discussed in Chapter 8. Conclusions and future research are discussed in Chapter 9.

Chapter 2
Background

In this chapter, an overview of program profiling and profile-guided optimization is given. Section 2.1 reviews the types of program profiles and the commonly used techniques to collect program profiles. In section 2.2 different profile-guided optimization techniques using these profiles are briefly reviewed.

2.1 Program profiles

Program profiles provide summary information on past program executions. In practice, *different* types of profiles are collected at *different* levels of granularity and used to guide *different* program optimizations.

2.1.1 Type of profiles

Three types of profiles are usually used in practice: control flow profiles, value profiles and address profiles.

Control flow profiles. Programs are usually represented by their control flow graphs (CFGs) during compiler analyses and optimizations. A control flow trace (CFT) remembers, in their execution order, all visited basic blocks in the CFG. By examining a CFT we can compute the execution frequency of any given program subpath. As expected CFTs can be extremely large in size and a number of approximations of CFT have been proposed and used to directly measure the execution frequencies of selected program subpaths. These profiles differ in the degree of approximation involved and the costs for collecting them. The proposed approximations of control flow profiles include the following:

- *Node profiles* provide the execution frequencies of the basic blocks in the control flow graph. Such profiles are adequate for some optimizations.

- *Edge profiles* provide the execution frequencies of each edge in the control flow graph. The overhead for collecting edge profiles is comparable to node profiles. However, edge profiles are superior to node profiles because edge profiles cannot always be computed from node profiles while node profiles can always be computed from edge profiles. Edges profiles are widely used.

- *Two-edge profiles* [36] provide the execution frequencies of each pair of consecutive edges in the control flow graph. Edge profiles can always be computed from two-edge profiles but the reverse is not true. Two-edge profiles derive their increased power from their ability to capture the correlation between the executions of consecutive conditional branches and they are used in a probabilistic data flow analysis framework [36] for computing frequencies of data flow facts.

- *Path profiles* [4] provide the execution frequencies of subpaths in the control flow graph that are acyclic and intraprocedural. Since a path is acyclic, it does not include a loop back edge and since it is intraprocedural, it terminates if an entry or an exit node of a procedure is reached. Path profiles are more precise than two-edge profiles for acyclic components of a control flow graph because they capture correlation across multiple conditional branches within an acyclic graph. However, two-edge profiles can capture correlation among a pair of conditional branches along a cyclic and interprocedural paths while path profiles cannot do so.

Since all of above profiles are approximations of original traces, some research has been done to evaluate how they differ from each other. Ball *et al.* [5] gave an algorithm to estimate the lower and upper bounds of path frequencies from edge profiles. Their results show that if a large percentage of a program's total flows are

definite, the estimated path frequencies from edge profiles can still identify *hot* paths. Otherwise more powerful path profiles should be used to identify *hot* paths.

Approximations are used in above profiles because the complete trace is large and it was believed to be too expensive to collect and use. This problem was addressed by Larus in [32]. He proposed to collect and compress the complete control flow graph trace using the Sequitur [40] algorithm. The compression result, identified as the *whole program path*, is a context free grammar that generates a single string which is the original control flow trace for the program. The redundancy in the original trace comes from frequently executed subpaths and it is removed by creating and reusing production rules.

Value profiles. Value profiles identify the specific values encountered as operands of an instruction and the frequencies with which these values are encountered. The example in Figure 2.1 illustrates the form of these profiles.

Code:

```
.....
I1: load R3, 0(R4)
I2: R2 ← R3 & 0xff
```

Value profile:

(instruction, register)	Profiles (value,freq)
(I1,R3)	(0xb8d003400,10) ...
...	...
(I1,R2)	(0,1000)
(I2,R3)	(0,100),(0x8900,200),...,(0x2900,100)

FIGURE 2.1. Value profiles.

Since the number of instructions in a program is large, and each operand of an instruction may potentially hold a very large number of values, collection of complete value profiles is not practical. Therefore to reduce the size of the profile data and the execution time overhead of profiling, the following two steps are taken.

First only the most frequently appearing N values are collected for a given operand. Calder et al.[11] have proposed maintaining a top-n-value table (TNV) for a register being written by an instruction. Each TNV table entry contains a pair of values:

the *value* and the *frequency* with which that value is encountered. A least frequently used (LFU) replacement policy is used to choose an entry for replacement when the table is full. If we exclusively use the LFU policy for updating the TNV, the values that are encountered later in the execution may not be able to reside in the table even if they are frequently encountered. This is because they may be repeatedly replaced. To avoid this situation, at regular intervals the bottom half of the table is cleared. By clearing part of the table, free entries are created that can be used by values encountered later in the program. Both the number of entries in the table and clearing interval are carefully tuned to get good results. Collecting only the top N values not only reduces the profiling overhead, but also makes convergence to a steady state faster.

The second complimentary approach in reducing profiling overhead is to collect value profiles for only interesting instructions. Watterson and Debray [58] use a cost-benefit model to identify interesting instructions. The cost is that to test whether a register has a special value; the benefit is the direct and indirect instruction savings that can be achieved by optimizing the program with this information. Control flow profiles are collected first to carry out cost-benefit analysis and to identify candidates for value profiles.

Address profiles. Address profiles can be collected in the form of a stream of memory addresses that are referenced by a program. These profiles are usually used to apply data layout and placement transformations for improving the performance of the memory hierarchy. Depending upon the optimization, the address traces can be collected at different levels of granularity. At the finest level of granularity, each memory address can be traced. Coarser level traces record references to individual objects rather than individual addresses.

A complete address trace of a program run can be extremely large. In order to compress the size of the address trace, Chilimbi [15] has proposed using the Sequitur

algorithm to generate a compressed whole program stream (WPS) representation of the address trace in much the same way as Sequitur is used to compress a program's control trace. To guide the application of data layout and placement transformations, the WPS representation is analyzed to identify hot address streams. These streams represent subsequences of addresses that are encountered very frequently during the program run.

Declarations:

```
int flag;
int *pa,*pb,*pc,*pd;
int buf[2000];
…
int xa,xb,xc,xd;
```

Sample code:

```
for(i=0;i<2000;i++) {
   swtich (flag) {
   case 1:
      xa = *pa;  … ; break;
   case 2:
      xb = *pb;  … ; break;
   case 3:
      xc = *pc;  … ; break;
   case 4:
      xd = *pd;  … ; break;
   }
   ….
   pa = buf[i]
   ….
}
```

Address profile:

Relationship	Profiles(frequency)
(A(xa),A(pa))	500
(A(xb),A(pb))	20
(A(xc),A(pc))	2
(A(xd),A(pd))	10
….	….
(A(pa),A(buf))	2000

FIGURE 2.2. Address profiles.

While the above approach first collects complete address profiles and then processes them to identify information useful in guiding data layout and placement transformations, another approach is to directly identify the useful information. Calder *et al.* [13] have proposed an algorithm based upon such an approach. The information that they collect is represented by a graph named the temporal relationship graph (TRG). The nodes in this graph are data items of interest. Weighted links are established between pairs of nodes. If references to a pair of data items are separated by fewer than a threshold number (say N) of other data references, then the weight associated with the link between the two items is incremented. To maintain the weights of all the links, an N-entry queue is maintained which records the latest

N data items that are referenced by the program. The weights on the links at the end of the program run can be used by the compiler to identify data items that should be placed close to each other for achieving good cache behavior. Figure 2.2 shows an example of the information collected using this approach.

2.1.2 Collecting profiles

Programs have to be executed in order to collect the program profiles. Three approaches are commonly used in practice for collecting profiles.

Instrumentation of the original program with new code to generate the profile data is the most widely used method. The introduced instrumentation code depends upon the types of profiles being collected. There are two possible ways to insert the instrumentation code. One way is to instrument at source or intermediate code level by modifying compilers [54, 55]. The instrumented source programs are then compiled normally to generate the executable code. The other way is to use a binary level instrumentation tool [19, 51] and insert the code directly into the executable code. While high level instrumentation can trace semantic information more easily, lower level instrumentation is sometimes easier to use and flexible.

The instrumented program is slower than the original version. While usually, the overhead of instrumented code is linear in the length of the execution, techniques have been proposed to reduce its overhead. Sarkar [48] proposed a technique to reduce the overhead in collecting control flow profiles. A counter is introduced for each control dependence region in the program; since they are far fewer than the basic blocks, the profiling overhead is reduced. Ball *et al.* [4] presented an algorithm to reduce the number of profiling points during the collection of path profiles.

Hardware profiling collects execution profiles with hardware support. Most modern processors [37, 26, 27] provide some hardware mechanisms for counting various types of dynamic information, such as cache misses, memory coherence operations,

branch mispredictions, and issued and committed instructions. MIPS R10000 [37] provides two 32-bit counters which can be used by the user to monitor 30 different events. Similarly, the event monitoring mechanism in the Intel Pentium 4 and Xeon processors [27] provides the flexibility to use 18 performance counters and to select 45 different events to be monitored. Hardware profiling is easy to use and incurs the least overhead. However, the counter based hardware profiling approaches lack the flexibility to monitor new events.

Simulation is another widely used approach in collecting and studying program profiles. It is especially important if we are studying the software and hardware interactions or if the target architecture does not exist. For example, Simplescalar [10], FAST [41] and RSIM [42] are cycle level architectural simulators; they provide ways to specify the features of simulated architectures. The advantage of this approach is that we can run the same program many times with different hardware configurations and study software and hardware interactions. The disadvantage is that it is very slow.

2.2 Profile guided optimizations

Different types of profiles are used to expose different optimization opportunities and assist in the development of different optimization techniques. These opportunities become available because of the dynamic inequality characteristics, e.g. some paths are executed more frequently than others, some variables are nearly constant, some data objects are referenced together, etc. A more precise cost-benefit model could be set up to evaluate this inequality and optimization transformations could thus be developed to generate more efficient code. This section reviews the optimization techniques proposed in the literature.

2.2.1 Profile guided control flow related optimizations

Control flow profiles are most widely used in optimization. Techniques are designed through *code specialization*, a technique that creates both optimized and unoptimized copies of statements and appropriate copy of the statement is executed depending upon the conditions that hold. Different code specialization algorithms are categorized primarily into two classes of transformations that are used to carry out code replication and enable specialization of conditionally optimizable code: code motion of different types and control flow restructuring with varying scope.

The basic form of code motion, namely safe code motion, in addition to honoring the program's data dependences, guarantees that for every execution of a statement during the execution of the optimized code, there exists a corresponding execution of the statement during the execution of the unoptimized code. As a consequence, it must be the case that if an exception occurs during the execution of optimized code, it would have also occurred during the original execution. Hardware support present in modern processors such as IA-64 [21] allows relaxation of the above constraint. In particular, *speculative code motion* allows the compiler to introduce executions of a statement in the optimized code that are not present in the unoptimized code. *Predicated code motion* [21] creates more opportunities by moving code out of control structures but still under correct predicates.

Control flow restructuring creates unoptimized and optimized copies of the statement and places them along the incoming edges. The primary cost in restructuring is the growth of code size. Control flow restructuring can be performed at different control flow granularities and scopes. Increasing the scope of restructuring also increases the growth of code size. Function inlining is one way to achieve interprocedural control flow restructuring. To limit code growth while performing interprocedural optimizations a couple of alternative techniques have been proposed: partial inlining of frequently executed paths through a procedure [25] and creating procedures with

multiple entries and multiple exits [6].

Existing transformations are enhanced and new transformations are developed to take advantage of profiles. They are used to develop a more precise cost-benefit model and estimate whether the benefit achieved from a particular transformation outweighs the cost that it introduced. For example, partial redundancy elimination (PRE) is traditionally performed using safe code motion [29]. The use of speculation was first proposed in [24, 23]. A control flow restructuring approach was proposed in [52]. A combination of all above transformations to achieve greater benefits at lower cost is discussed in [7].

2.2.2 Profile guided value optimizations

Value profiles can be used to identify almost invariant variables for constant folding, strength reduction, code specialization, adaptive execution and guiding dynamic compilation.

Muth, Watterson and Debray [39] introduced a value profile based code specialization technique which has in three steps. First, using basic block profiles, program points and registers are identified where specialization might be profitable. Second value and expression profiles are obtained for these program points. Third, these collected profiles are used to carry out specialization for those program points that are deemed profitable.

Dynamic optimization [3] and adaptive execution [28] generate specialized code either from scratch or from a statically generated template. Value profiles can help to identify the semi-invariant variables statically and reduce greatly the optimization cost at runtime.

Calder and Feller *et al.* [12] discussed different computer architecture components that can benefit from value profiles. Hardware value predictors [34], for example, can benefit in several ways from value profiles. By classifying instructions into pre-

dictable, not predictable, or hard to predict, one can determine which instructions to statically predict or not to predict. Value profiling can even be used to classify instructions indicating which type of predictor would better predict the instruction in a hybrid predictor. This increases the prediction accuracy and decreases the conflicts or aliasing in a prediction table.

2.2.3 Profile guided memory optimizations

Over the past decade, while the processor speed have risen by 55% each year, the memory speeds have only improved by 7% each year. As a result, the memory becomes a major bottleneck in performance improvement and so has drawn a lot of attention. The techniques proposed to optimize memory performance span a wide range of categories.

- **Object placement.** This type of technique determines a better placement scheme of data objects to improve cache behavior. *Memory forwarding* proposed by Luk and Mowry [35] attached one bit to each word in the memory. An object can be migrated dynamically according to its runtime behavior. After its migration, the memory address where it previously resided saves an indirect pointer to the new address. The additional bit is set to indicate that the object has moved. Other approaches try to place an object in a desired places. *Ccmalloc* [17] for example enhanced the system memory allocator by one more parameter used as its parent pointer. Whenever possible, the new object is placed into the same cache block as the existing object. The address profiles can be used to identify objects that are accessed contemporaneously.

- **Object layout.** This type of technique determines a layout of fields within a large data object to improve cache locality. A data structure is often defined by the programmer to support code readability. The compiler simply uses a memory layout for the fields which mirror the order they are declared. However,

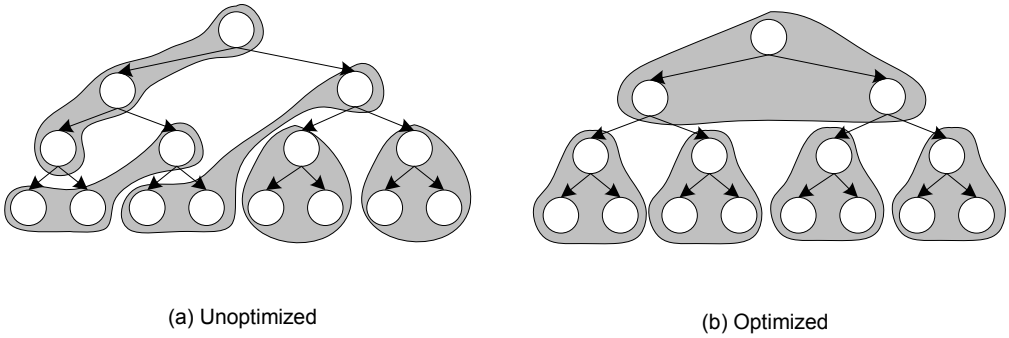


FIGURE 2.3. A ccmalloc example.

this order may not be consistent with the order that incurs fewer cache misses. Truong *et al.* [56] evaluated an approach to reorder the fields and showed that a node that spans several cache blocks can take advantage of cache line prefetching and reduce cache pollution, thus improving cache performance.

- **Hybrid scheme.** This type of technique combines the object placement and layout approaches to further improve the performance. In [16], object splitting technique was proposed to split an object into two parts: the hot primary part and the cold secondary part. Hot fields are accessed directly while the cold ones are accessed through a pointer stored in the hot part. Locality is improved by reducing the data object size and benefits most memory accesses for hot fields.

Chapter 3
Compressing the control flow trace

A control flow trace is a sequence of basic block instances in their execution order. Node, edge or path profiles can be viewed as lossy compressed representations of the control flow trace. Until recently, it was believed that a complete control flow trace is too expensive to collect and use. However, Larus [32] recently demonstrated that it is feasible to effectively collect a whole program path (WPP), which is the compressed form of a complete control flow trace. By using the Sequitur [40] algorithm, Larus showed that a control flow trace which is typically very large (100's of MBytes), can be compressed (10's of MBytes) and saved for future analysis.

While the compression algorithm proposed by Larus is highly effective, the compression is accompanied with a loss in ease of accessibility to information. For example, path traces pertaining to a particular function cannot generally be obtained without examining the entire compressed WPP representation. This is a serious drawback because typically an application using the WPP can be expected to make a series of requests for profile data for individual functions, that is, each request asks only for a small subset of the overall information contained in a WPP. Repeated extraction operations to satisfy these requests are likely to result in high analysis time costs. Therefore it is important to design a representation from which path traces of individual functions can be rapidly accessed.

The above loss of accessibility is a natural consequence of of treating the entire control flow trace as a single data stream during *compression*. As a result the information corresponding to a given function is scattered throughout the compressed trace and can in general be located only by examining the entire compressed trace.

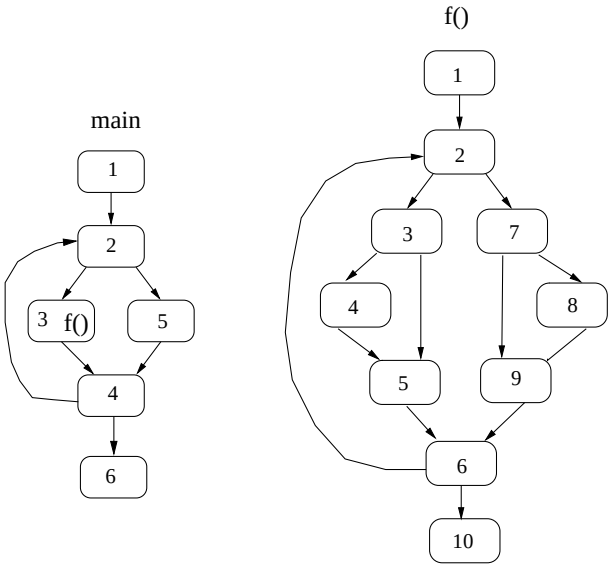
In order to solve this problem a new compression approach is proposed in this

dissertation which aims at simultaneously reducing the size of the control flow trace and providing easy access to subsets of information within the compressed trace. The approach organizes the information contained in a complete trace as follows. The control flow trace is first broken into *path traces* corresponding to individual function calls, and all of the path traces for a given function are stored together as a block. Therefore information regarding a specific function can be readily accessed. In order to ensure that the complete control flow can be reconstructed from individual path traces, a *dynamic call graph* which links the *path traces* together is also maintained. The detailed compression algorithm for control flow traces is presented in this chapter.

The rest of this chapter is organized as follows. Section 3.1 introduces the new timestamped whole program path (TWPP) representation. The algorithm steps are given to convert a control flow trace into the final representation. Section 3.2 presents the experimental results, comparing the compression ratio as well as the access time using different compression algorithms. Section 3.3 summarizes the chapter.

3.1 TWPP: Timestamped whole program path

As mentioned earlier, a whole program path (WPP) is the compressed form of a control flow trace from a program execution. Consider the program and a sample control flow trace shown in Figure 3.1. The trace shows that the loop in `main` iterates 5 times and in each iteration the function `f` is called. The loop in function `f` iterates 3 times for each call. Looking at the WPP for a small program we observe two things: WPPs for real applications can be expected to be quite large (e.g., 100's of MBytes) and in its current linear form WPP is difficult to use (e.g., in order to extract trace information for a subpath in `main` or function `f`, we must examine the entire WPP). Next we present a step by step transformation of the above WPP to achieve two goals: compaction of the WPP to reduce memory requirements and organization of the WPP information for faster access to path traces of individual functions.



main(1.2.3.f(1.2.7.8.9.6.2.7.8.9.6.2.7.8.9.6.10).4.
2.3.f(1.2.7.8.9.6.2.7.8.9.6.2.7.8.9.6.10).4.
2.3.f(1.2.3.4.5.6.2.3.4.5.6.2.3.4.5.6.10).4.
2.3.f(1.2.7.8.9.6.2.7.8.9.6.2.7.8.9.6.10).4.
2.3.f(1.2.3.4.5.6.2.3.4.5.6.2.3.4.5.6.10).4.6)

FIGURE 3.1. An uncompacted control flow trace.

Partitioning WPP into path traces. We partition the WPP into *path traces* corresponding to individual function calls and all of the path traces for a given function are stored together as a block. Therefore information regarding a specific function can be readily accessed. In order to ensure that the complete WPP can be reconstructed from individual path traces, a *dynamic call graph* (DCG) which links the *path traces* together is also maintained. Figure 3.2 shows this representation of the WPP for our example program. Clearly from this representation the WPP form of Figure 3.1 can be easily constructed. More importantly one can rapidly search for occurrences of a given path (intraprocedural or interprocedural). The path traces of interest are located and then examined for desired information. To search for an occurrence of a path in `main` we need to only examine one-sixth of the total trace in Figure 3.2.

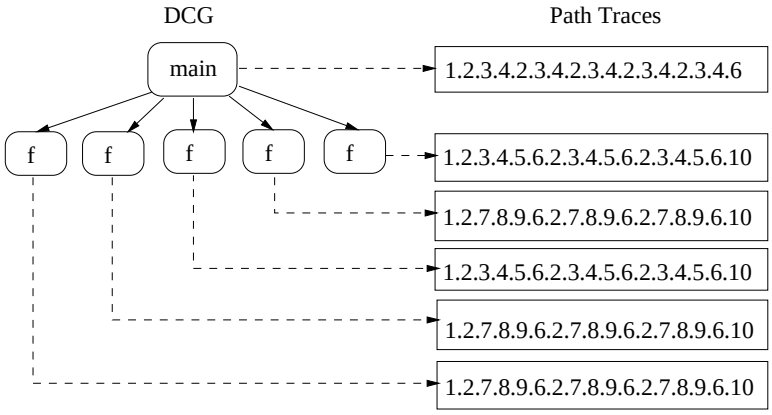


FIGURE 3.2. WPP organized using the DCG.

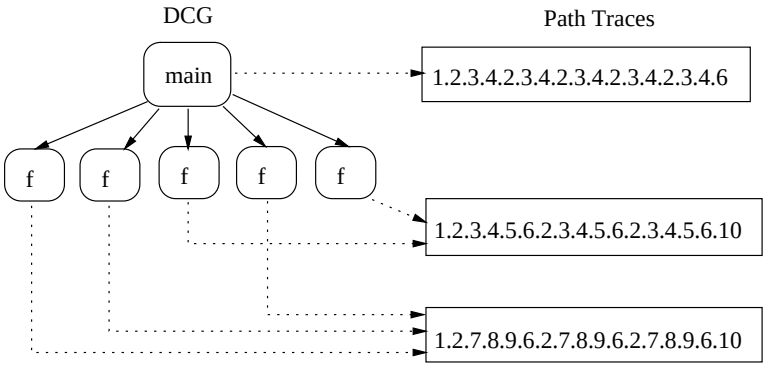


FIGURE 3.3. WPP after redundant path trace removal.

Eliminating redundant path traces. The WPP can be greatly reduced in size by *eliminating duplicate path traces* generated by different calls to the same function. In Figure 3.2, corresponding to the 5 calls to function `f`, there are only two unique path traces. Therefore the WPP representation can be transformed to eliminate redundant path traces as shown in Figure 3.3. This technique is very effective because although many functions are called numerous times, they tend to follow one of a small subset of paths through the function body. For example, in a WPP collected from executing `gcc` we found that function `_rtx_equal_p` was called 355189 times but it generated only 35 unique path traces.

Creating dictionaries of dynamic basic blocks. Another technique that we employ replaces a sequence of static basic block ids that correspond to a *dynamic basic block* by a single id. A *dynamic basic block* (DBB) belonging to a path trace is a sequence of static basic blocks that is always entered from the first block and exited from the last block in the path trace. Since DBBs can often appear inside loops, they are often repeated many times in a path trace. Thus, replacing them by a single id can significantly reduce the size of the WPP.

Each path trace is processed as follows: a dictionary of DBBs is created by constructing a dynamic control flow graph and finding chains of static blocks representing DBBs in it. Each DBB is assigned the block id of the first static block in it and accordingly the path trace is modified by deleting all but the first id in each occurrence of a DBB. Once all compacted path traces and dictionaries are obtained, duplicate path traces and dictionaries are also eliminated. In this transformed form, each node in the dynamic call graph has an associated tuple (t, d) where t is a path trace and d is a dictionary. Figure 3.4 shows the chains of static basic blocks that form dynamic basic blocks for the three path traces in Figure 3.3. After creating dictionaries and compacting path traces, we are left with one path trace and two dictionaries for function `f` as shown in Figure 3.5.

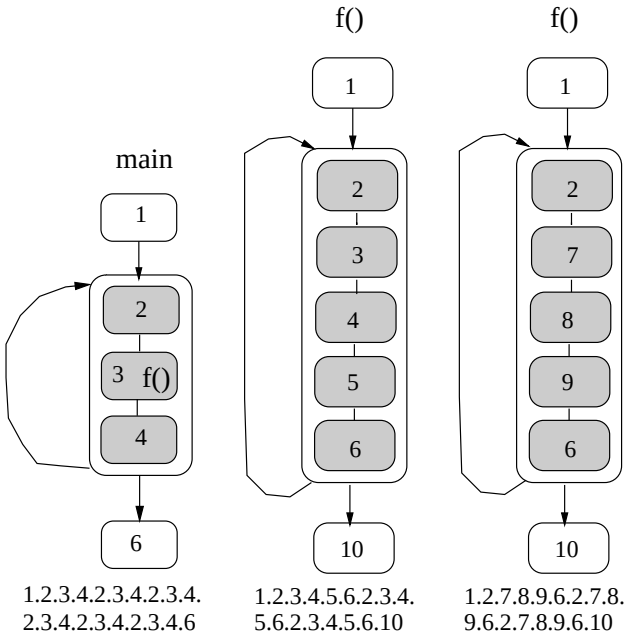


FIGURE 3.4. DBBs and dynamic control flow graphs.

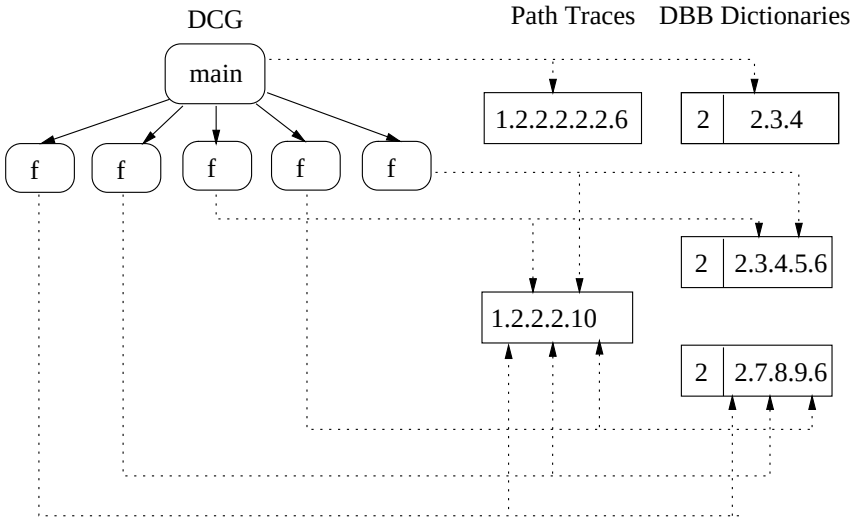


FIGURE 3.5. WPP after creating dictionaries of DBBs.

Timestamped WPP representation. In the WPP representation described so far, the execution trace of a given function invocation is represented by a sequence of basic blocks visited during its execution. While such a path trace representation is adequate for identifying hot paths through a program, it is not the most appropriate for performing data flow analysis. Since profile-limited data flow analysis is carried out from the perspective of basic blocks, it is more appropriate to organize the traces from the perspective of dynamic basic blocks. Next we describe the *timestamped* WPP (TWPP) representation which achieves this goal.

The execution of the function can be viewed from the perspective of time steps, where each time step corresponds to the execution of a dynamic basic block. Therefore a path trace for a function call in a WPP representation can be viewed as a mapping between time steps, or *timestamps*, and dynamic basic blocks. In contrast, the TWPPs represent a mapping between dynamic basic blocks and an ordered sets of timestamps. Let $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{P}(\mathcal{T})$ denote the set of timestamps, set of dynamic basic blocks, and the power set of timestamps associated with the path trace of a given function call f. A path trace in WPP and TWPP forms is represented by the following mappings:

$$\mathbf{W}PPPathTrace_f : \mathcal{T} \rightarrow \mathcal{B}$$

$$\mathbf{T}WPPPathTrace_f : \mathcal{B} \rightarrow \mathcal{P}(\mathcal{T})$$

Consider the WPP of Figure 3.5. The WPP trace 1.2.2.2.2.2.6 corresponds to the following $\mathcal{T} \rightarrow \mathcal{B}$ mapping: $\{1 \rightarrow 2,\ 2 \rightarrow 2,\ 3 \rightarrow 2,\ 4 \rightarrow 2,\ 5 \rightarrow 2,\ 6 \rightarrow 2,\ 7 \rightarrow 6\}$. When transformed to TWPP form it is represented by the following $\mathcal{B} \rightarrow \mathcal{P}(\mathcal{T})$ mapping: $\{1 \rightarrow \{1\},\ 2 \rightarrow \{2,3,4,5,6\},\ 6 \rightarrow \{7\}\}$. The complete uncompacted TWPP for this example is shown in Figure 3.6.

Compacting TWPP path traces. The path traces in TWPP form can be further compacted because often a subsequence of timestamp values corresponding a

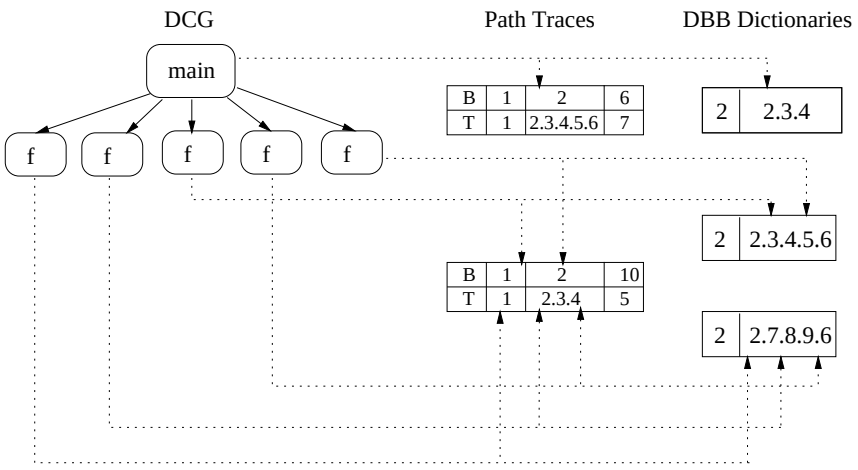


FIGURE 3.6. TWPP form.

dynamic basic block forms an arithmetic series. This situation arises particularly when the same path within a loop body is traversed repeatedly during different loop iterations. The subsequences that form arithmetic series are compacted yielding a sequence of entries which are of the following form: l (singleton), $l : h$ ($l.l + 1.l + 2...h$, i.e., series with step 1), or $l : h : s$ ($l.l + s.l + 2s...h$, i.e., series with step s). As we can see, depending upon its form, an entry is represented using one, two or three positive integer values. We store the timestamps corresponding to a block merely as a sequence of integers. For correct interpretation of the information we need to encode the boundaries that separate the variable length entries. This information is encoded in the signs (+ or -) of the values and therefore it does not require any increase in the size of the path trace. In particular, the last number in a each entry is stored as a negative number. By using the sign to encode the end of an entry we limit the largest timestamp value that is available to us since we can no longer use unsigned integers. However, our experience with the benchmarks considered shows that the timestamp value does not overflow because individual path traces are much smaller than the complete WPP.

Notice that the sequence of timestamps assigned to dynamic basic block 2 in Figure 3.6 form an arithmetic series since block 2 is executed repeatedly during suc-

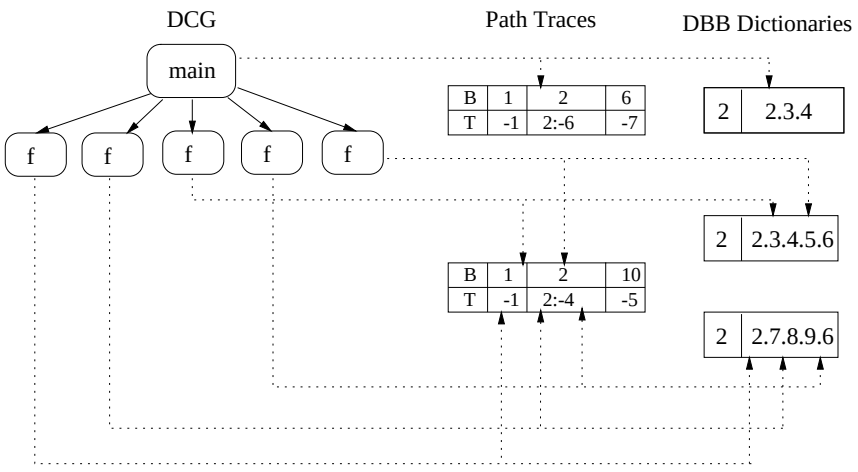
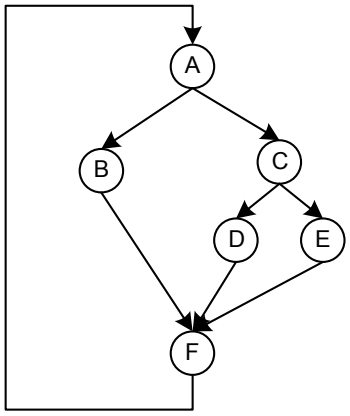


FIGURE 3.7. Compacted TWPP.

cessive loop iterations. Therefore the TWPP can be compacted into: $\{1 \rightarrow \{-1\}$, $2 \rightarrow \{2 : -6\}, 6 \rightarrow \{-7\}\}$. Notice that the last number in each sequence is a negative number. The complete compacted form of TWPP for our running example is shown in Figure 3.7.

It is also possible to increase the compressibility of timestamps associated with basic blocks using a simple technique. Consider a situation in which different paths through a loop body of a function contain different numbers of nodes. For example, in Figure 3.8, there are three paths from A to F: paths ACDF and ACEF contain three nodes while ABF contains 2 nodes. Even though nodes A and F are executed along each of these paths, their timestamps are irregular due to the different number of nodes along the paths. However, if all paths contained the same number of nodes, then no matter which path is taken during each loop iteration, the nodes A and F would have had perfectly compressible series of timestamps. To address this problem we associated weights to edges where the weights are used to generate timestamps. In particular, the weight of an edge represents the amount by which the timestamp is incremented when the edge is traversed. By assigning weights to edges such that sum of weights of edges along each of the paths through the loop is the same, we can



Trace:
ACDF ABF ACEF
ACEF ABF ACDF
ABF

Edge Unbalanced TWPP

Node	Timestamp
A	1:-5,8:16:-4,19:-23
B	6:-17,-24
C	2:-9,13:-20
D	3:-21
E	10:-14
F	4:-7,11:-15,18:-22,-25

Edge Balanced TWPP (W_{BF} = 2)

Node	Timestamp
A	1:25:-4
B	6:-18,-26
C	2:-10,14:-22
D	3:-23
E	11:-15
F	4:28:-4

FIGURE 3.8. Balancing example.

guarantee that nodes that are visited along each path have compressible timestamps. In our example the edge BF is assigned the weight 2 while all the other edges are assigned the weight of 1 in order to balance the paths. The result is that the timestamps for nodes A and F can now be compressed and the size of TWPP representation of nodes A and F is further reduced.

Compacting the DCG. The dynamic call graphs resulting from executions of large application programs can also be quite large. Therefore in addition to compacting the path traces, we also compress the DCG. For this purpose we considered the popular dictionary based approaches proposed by Ziv and Lempel [65, 66]. In particular, we used Welch's variation of Ziv and Lempel's adaptive dictionary based technique which is also referred to as the LZW algorithm [60].

3.2 Implementation and experiments

The TWPP algorithm has been implemented and evaluated to compact whole program paths for several benchmark programs from the SPECint95 suite [50]. The original WPPs used in the experiments were generated using the Trimaran compiler infrastructure [55]. A WPP consists of two parts: the dynamic call graph (DCG) and the individual traces for function calls (which are collectively referred to as the WPP traces). The sizes of WPPs used in the experiments are shown in Table 3.1. The experiments are aimed at studying the effectiveness of our compaction techniques in reducing memory requirements and the effectiveness of organization of the WPP information for faster access.

Program	DCG (MB)	WPP traces (MB)	Total size (MB)
`099.go`	6.0	170.0	176.0
`126.gcc`	34.7	489.5	524.2
`130.li`	6.6	78.3	84.9
`132.ijpeg`	1.7	266.9	268.6
`134.perl`	3.4	41.5	44.9

TABLE 3.1. Sample input traces used in the experiments.

Program	WPP trace after		Compacted TWPP trace - MB	OWPP / CTWPP
	Redundancy removal - MB	Dictionary creation - MB		
`099.go`	27.0 (x6.30)	17.1 (x1.58)	17.6 (x0.97)	9.7
`126.gcc`	86.5 (x5.66)	50.8 (x1.70)	32.9 (x1.54)	14.9
`130.li`	8.5 (x9.21)	5.3 (x1.60)	1.1 (x4.81)	71.2
`132.ijpeg`	28.1 (x9.50)	20.8 (x1.35)	5.7 (x3.65)	46.8
`134.perl`	7.2 (x5.76)	1.7 (x4.24)	0.02 (x85.0)	2075

TABLE 3.2. WPP trace compaction due to various transformations.

Program	Compacted DCG (MB)	Compacted TWPP (MB)		Total (MB)	Compaction factor
		Traces	Dictionaries		
099.go	6.6	17.6	2.3	26.5	7
126.gcc	13.8	32.9	4.9	51.6	10
130.li	5.3	1.1	0.04	6.4	13
132.ijpeg	1.0	5.7	0.6	7.3	37
134.perl	0.7	0.02	0.02	0.7	64

TABLE 3.3. Overall compaction factor.

Compaction study. Table 3.2 shows the sizes of the WPP traces in their various forms. As we can see, the three compacting transformations, removal of redundant path traces, creation of DBB dictionaries, and transformation to compacted TWPP form are all very effective in reducing the WPP trace size. The ratio of the sizes of original WPP traces (OWPP) and compacted TWPP traces (CTWPP) gives us the compression factor which varies from 9.7 to 2075 for our sample traces. The sizes of the WPP traces after each of the three transformations as well as the compression factors corresponding to each of the transformations are also shown separately in parenthesis in Table 3.2. The results show that each of the transformations is an important source of compaction.

A large factor of size reduction comes from removing redundant path traces (5.66 - 9.50). The reason for this large reduction becomes clear when the data in Figure 5.1 is examined. This figure gives the percentage of total function calls (plotted along Y-axis) that can be attributed to functions with at most N unique path traces (N is plotted along the X-axis). For `130.li`, `132.ijpeg`, and `134.perl` programs 57-80% of all function calls are attributable to functions with at most 5 unique path traces. For `126.gcc` and `099.go` over 50% of function calls are attributable to functions with at most 25 and 50 unique traces respectively. Given that the number of function calls made during the runs of these benchmarks were in hundreds of thousands, we can see that the degree of redundancy in path traces is very high.

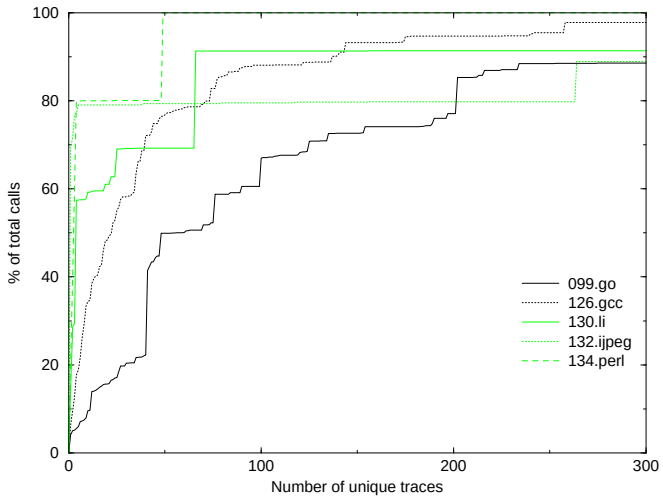


FIGURE 3.9. Trace redundancy.

The creation of dictionaries results in compaction of WPP traces by factors ranging from 1.35 to 4.24. The conversion into compacted TWPP form results in further reductions. For four out of five benchmarks, compacted TWPP traces provide reductions in the sizes of WPP traces by factors ranging from 1.54 to 85. The only case in which compacted TWPP trace is slightly larger is the `099.go` program where the compacted TWPP trace was 3% larger than the compacted WPP trace prior to its conversion to TWPP form. These results are very encouraging because not only is the TWPP representation suitable for profile-limited data flow analysis, it is also compact.

The compacted sizes at different algorithm steps are plotted in Figure 3.10 which gives a visual comparison. The step to eliminate redundant path traces is very effective and the step to convert to TWPP representation also contributes a lot to the compression.

The breakdown of different components of a WPP and the overall compaction factors for the complete WPP (DCG + WPP trace) are given in the Table 3.3. For the sample WPPs used in these experiments the overall WPP compaction factor ranges from 7 to 64.

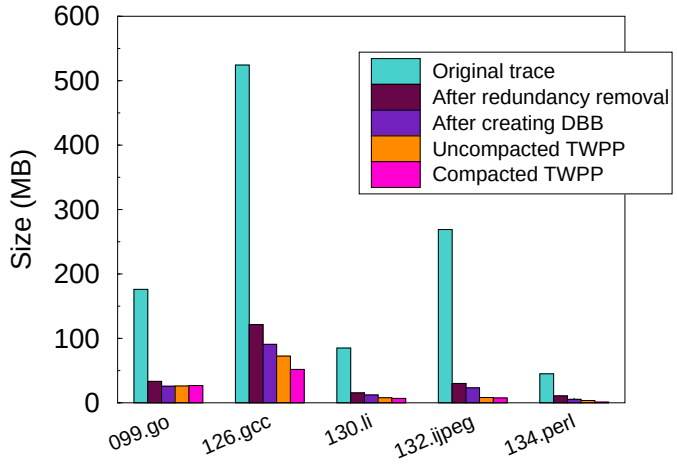


FIGURE 3.10. Compacted size at each TWPP step.

Access time study. To study the impact of reductions in the WPP size on the speed with which the path traces can be accessed, an experiment was conducted which measured the time it took to extract the path traces corresponding to a single function from the complete WPP. The expected speedups result from two sources. First due to the compaction of the WPP we have to read through a smaller file. Second the contents of the file are organized to allow faster access. Followed by the dynamic call graph, the path traces (including dictionaries) of the most frequently called function are stored first and that of least frequently called function are stored last. By remembering the position of information for each function in the file, and storing it as a header in the compacted TWPP file, the path traces for individual functions are rapidly accessed.

Table 3.4 shows the times taken to extract a function's trace in the following scenarios: extraction from uncompacted file (column U); and extraction from compacted file (column C). Both the average and maximum times for U and C are given. On average the access times are reduced by over 3 orders of magnitude.

Larus's Sequitur based compression algorithm. For comparison Larus's compression algorithm which is based upon Sequitur [40] was also implemented. This

Program	U (ms)		C (ms)		Speedup
	avg.	max	avg.	max	C/U(avg.)
099.go	5033	8383	8	1438	629
126.gcc	22879	29672	6	528	3813
132.ijpeg	7615	11447	6	258	1269
130.li	2390	3263	2	124	1195
134.perl	1303	1873	0.2	3	434

TABLE 3.4. Extraction times for a single function.

algorithm produces the compressed WPP representation which is in the form of a grammar that generates a single string - the original trace. The Sequitur generated grammar representation was compared with the TWPPs generated in two ways: their sizes and the access times to individual function traces.

The results of this comparison are shown in Table 3.5. On average, the total size of the grammar produced by Sequitur is smaller than the corresponding size of the compacted TWPP by a factor of 3.92. Now consider the time it takes to extract the trace corresponding to a single function from the complete compacted trace. The extraction of a function's trace from the Sequitur generated grammar essentially requires two steps: reading in the grammar and then processing it to generate a subgrammar corresponding to the functions trace. The total time taken for extraction, and the times for each of the steps, are shown in Table 3.5. These numbers represent averages over all functions present in the respective programs. These times range from 10's to 1000's of milliseconds. In contrast, the TWPP is so organized that we can locate and extract the trace in a few (< 10) milliseconds. The access times for Sequitur grammars are greater than access times of TWPPs by factors ranging from 89 to 553. In summary, although TWPPs are larger in size by an average factor of 3.92, they provide access times that are lower by an average factor of 309. These experiments show that the two representations embody design decisions with different space time trade-offs.

Program	Compacted size		Extraction time	
	Sequitur (MB)	TWPP (MB)	Sequitur (ms) read+process=total	TWPP (ms)
099.go	8.4	26.5	622 + 1315 = 1937	8
126.gcc	11.2	51.6	898 + 2423 = 3321	6
132.ijpeg	0.7	6.4	544 + 1650 = 2194	6
130.li	7.8	7.3	47 + 132 = 179	2
134.perl	0.4	0.7	29 + 30 = 59	0.2

TABLE 3.5. Compacted trace sizes and extraction times.

Apart from the different size and access time characteristics, the two representations also impact on the design of analysis algorithms that will use them. While Larus's techniques is suitable for analysis of hot paths (i.e., collection of data flow facts that hold along frequently executed paths), TWPP representation is suitable for collecting hot data flow facts (data flow facts that hold frequently at various program points). One of the advantages of our approach is that TWPPs are in the form required for profile-limited analysis. In contrast the compressed WPPs produced by Sequitur require some amount of preprocessing before they can be used by an application.

3.3 Conclusion

A new timestamped whole program path representation is proposed in this chapter to compress the complete control flow trace. Without compromising accessibility, it achieves effective size compaction. The organization of the trace information based upon the dynamic call graph and timestamped dynamic basic blocks is particularly appropriate for performing fast accesses to path traces of a given function. It compacts the original traces by factors ranging from 7 to 64 and at the same time speedups of over 3 orders of magnitude were observed in responses to queries requesting all of the trace information of a given function.

Chapter 4
Compressing the memory dependence trace

In the preceding chapter, a new timestamped whole program path (TWPP) representation was proposed to compress a complete control flow trace. Compared to the original WPP which is compressed by Sequitur, TWPP is more suitable for data flow analysis. However, control flow information alone is not sufficient for inferring certain dynamic data flow facts. If the data flow fact is related to dynamically allocated data structures or depends on the dynamic memory accesses then control flow information alone is insufficient. For example, Figure 4.1(a) shows the control flow graph of a small program in which the results from a simple function and a complex function are assigned to address "p" and "q" respectively. If "p" and "q" almost always point to the same address, it would be beneficial to transform the code to Figure 4.1(b) in which the complex function is called only when the result will not be overwritten later. However, if "p" and "q" rarely point to the same address, this transformation should not be applied since the overhead introduced due to the additional comparison would outweigh the benefits.

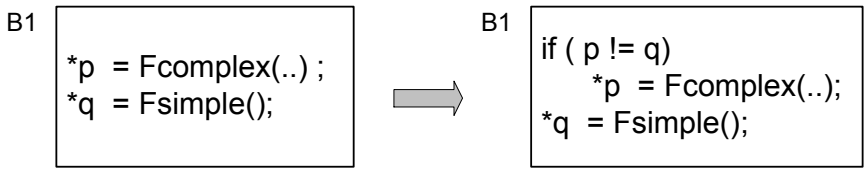


Figure 4.1. Importance of data flow information.

The precise dynamic data flow consists of two types of flows: those accessed through registers and those accessed through memory locations. The compiler will

translate a high level variable access into one of these two types. Since most instructions have two or three operands, the size of a precise dynamic data flow tracing would be about two to three times the execution length. Luckily, the data flows through registers are easy to determine, since registers are always accessed through their explicit names and therefore the data flow through registers can be reconstructed easily from its control flow. Only the data flows through memory locations are implicit, since the location is decided by the runtime value of the source address register of a load or the value of the destination address register of a store. Thus, to precisely trace the dynamic data flow, memory addresses in the memory access instructions, i.e. the values in the above discussed registers, are traced and saved into a file.

Usually memory address traces are much bigger than control flow traces. It is harder to compress a memory address trace than it is to compress a control flow trace. One reason is that the number of accessed memory addresses is larger than the total number of basic blocks that a program can have. Most compression algorithms, including Sequitur, are less effective in handling a stream of text over a huge alphabet and thus do not achieve very good compression ratios in compressing a memory address trace. Moreover, a basic block id is usually represented by a half-word (16 bits or less), while a memory address is represented by a whole word (32 bits).

This chapter will enhance the timestamped representation from the preceding chapter to compress memory address traces with the aim of providing data dependence information during data flow analysis. The rest of this chapter is organized as follows. Section 4.1 discusses the compression steps in constructing the enhanced timestamped whole program path (TWPP+). The implementation and experimental results are shown in section 4.2. Section 4.3 discusses the related work. Section 4.4 concludes the chapter.

4.1 TWPP+: TWPP with memory dependence edges

Given a whole program trace in which the basic block ids as well as memory addresses accessed within each basic block are represented in their execution order, the enhanced TWPP representation first separates the control flow trace and the memory address trace. The control flow trace is represented by the TWPP representation introduced in the preceding chapter. The memory address trace is handled as described in the following steps.

Eliminating explicit memory addresses. As discussed above, the purpose of including the memory address trace in a whole program path profile is to reconstruct the precise and complete data flows for a given execution. In such a scenario, the absolute addresses themselves are not important – only the load/store dependences instead. Thus the first step is to eliminate the explicit addresses and remember only dynamic data flows.

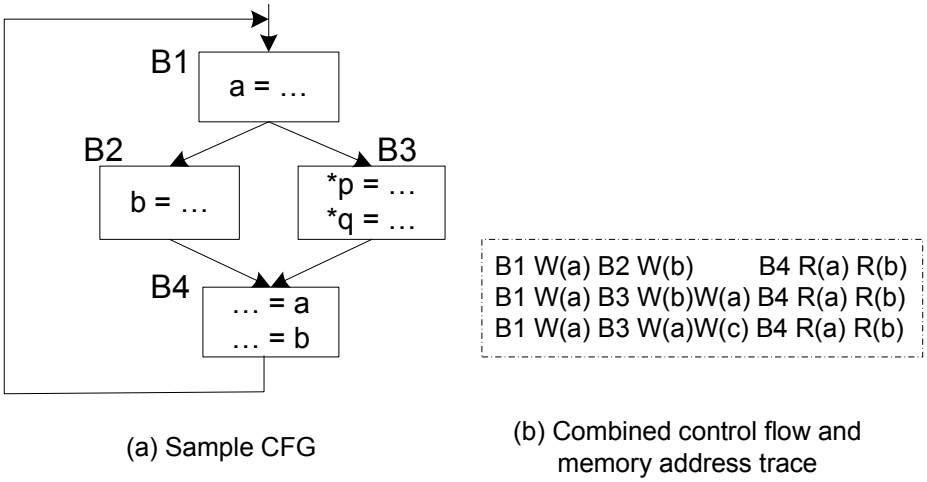


FIGURE 4.2. An example of a memory dependence trace.

A data flow through a memory location exists from an instance of a store instruction to an instance of a load instruction, denoted by a memory dependence edge from the store instance to the load instance. To precisely remember a dynamic data flow

edge, both the load and store instruction ids as well as their instance counts need to be remembered. For example, in Figure 4.2, B_3 was executed twice during which it wrote to locations for "b,a" and "a,c" respectively. If we want to set up the data flow dependence for the load of "b" at the end, we need to indicate that it gets the value from the *first* execution instance of basic block B_3 and it is the *first* memory access instruction in this basic block. Similarly, a function instance count needs to be remembered as the function might be executed several times. Thus, a dynamic memory dependence edge is precisely defined by

$$[< (F_0, FC_0), (B_0, BC_0), S_0 >, < (F_1, FC_1), (B_1, BC_1), S_1 >]$$

where F, FC, B, BC, S denote respectively the function id, the call instance count of the function, the basic block id, the instance count of the basic block, the sequence index of the memory access inside a basic block.

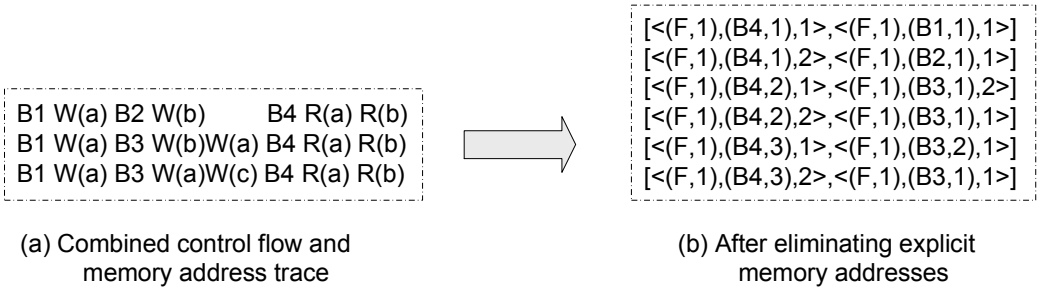


FIGURE 4.3. Eliminating explicit addresses.

For the sample trace shown in Figure 4.2, after eliminating the explicit addresses, it can be represented by explicit memory dependence edges shown in Figure 4.3(b).

Identifying the appropriate set to represent. As it was discussed, the data flows that pass through registers need not traced because they could be reconstructed from the program's control flow trace. Similarly, a data flow, even if it passes through a memory location, could be safely discarded if it can be reconstructed from the program's control flow. Since all data flows are represented as memory dependence edges

after the first step, an edge could be discarded if the corresponding data flow has this property. For example, in Figure 4.2, B_2 writes to "b" and the second access of B_4 reads "b". If B_2 is executed before B_4, there must be a data dependence between them. This edge $< F, 1, B_4, 1, 2 >, < F, 1, B_2, 1, 1 >$ can be constructed from the control flow. On the other hand, the edge $< F, 1, B_4, 3, 2 >, < F, 1, B_3, 1, 1 >$ cannot be eliminated because the first store instruction of B_3 may or may not write to the address "b". This edge cannot be recovered from the control flow. Thus, a memory dependence edge can be removed only if both its load instruction and its store instruction access only one memory address and this address is statically decidable.

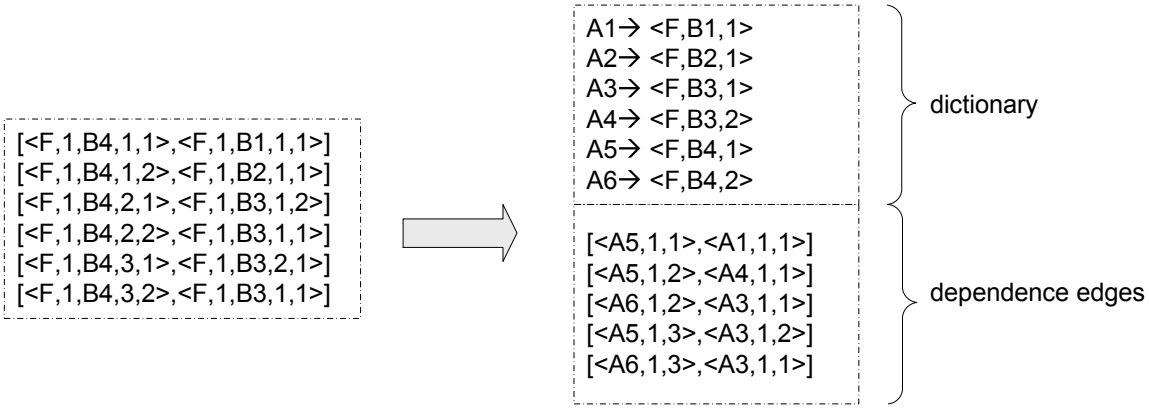


FIGURE 4.4. Creating memory access dictionary.

Another type of redundancy exists in the representation of each data dependence edge. Each edge consists of two instruction instances (see Figure 4.3) and each instance is decided by 5 items: the function id, the basic block id of the function, the instruction index of the basic block, the function instance count and the basic block instance count. The first three items are statically decidable and thus could be combined. We globally number all load and store instructions and each instruction is assigned a unique id. Now, an edge can be represented by

$$[< G_0, FC_0, BC_0 >, < G_1, FC_1, BC_1 >]$$

where G represents the global unique memory access point, i.e. a load or store

instruction point. There is a global mapping from each global unique id to a triple that consists of the function id, the basic block id and the instruction index in the basic block. This mapping is identified as *memory access point dictionary* and is stored aside for future reference.

Figure 4.4 shows the result after removing control flow decidable edges and creating the memory access point dictionary.

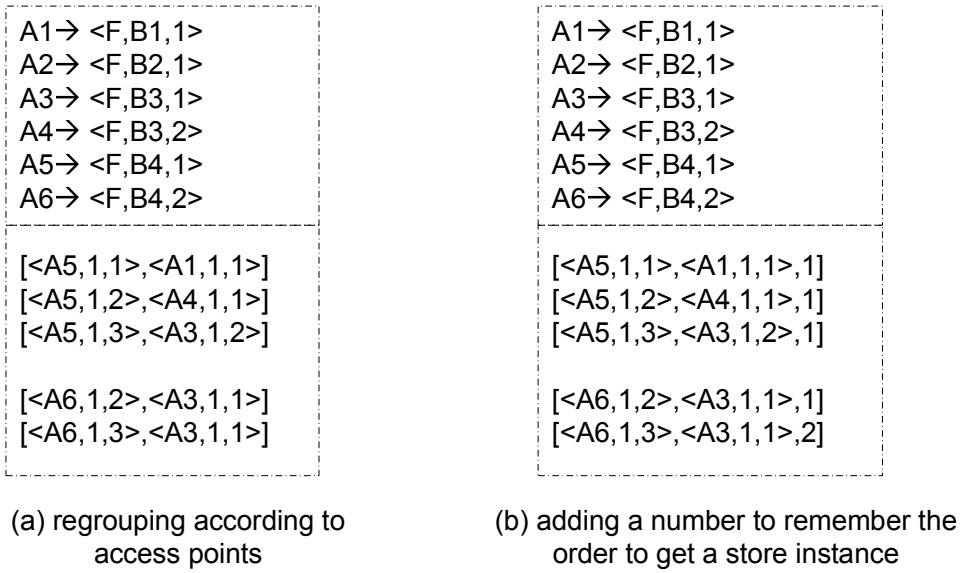


FIGURE 4.5. Representation for timestamped memory dependence edges.

Timestamped memory dependence edges representation. In the TWPP representation, a function level path is converted to a sequence of timestamps at which it is executed. Similarly, the sequence of dependence edges are regrouped according to their global unique ids. There are three reasons why it is organized in this way rather than grouping them according to individual function calls. First, data dependence edges are not directly connected with the control flow. Distinct instances of the same static load instruction might be dependent upon different store instruction instances within the function or even from other functions. Second, each function has many call instances. Unlike functional level control flow traces, it is less likely that the edge se-

quences from two call instances are exactly the same. Grouping edges at the function level will not lead to the discovery of much redundancy. Third, memory dependence edges show significant repetition at the same memory access point. Organizing edges according to memory access points can increase compression opportunities.

After grouping, the relative order of the edges is lost. Although it is possible to rebuild the order by employing the control flow trace, it is generally too expensive. Usually, the order between two edges is of interest if they share the same load or store instruction. If they share the same load instruction, they are grouped in the same block, and the relative order is determined by their function instance and basic block instance counts. If they share the same store instruction, their relative order could be easily determined if they are from different instances. However, in order to distinguish those that share the same store instance, we remember one additional number i for each edge which indicates this load instance is the i-th load from that store instance.

The result after grouping for the previous example is shown in Figure 4.5(a). The result after adding the extra sequence number is shown in Figure 4.5(b). Three edges at the fifth access point are organized as size sets of grammars.

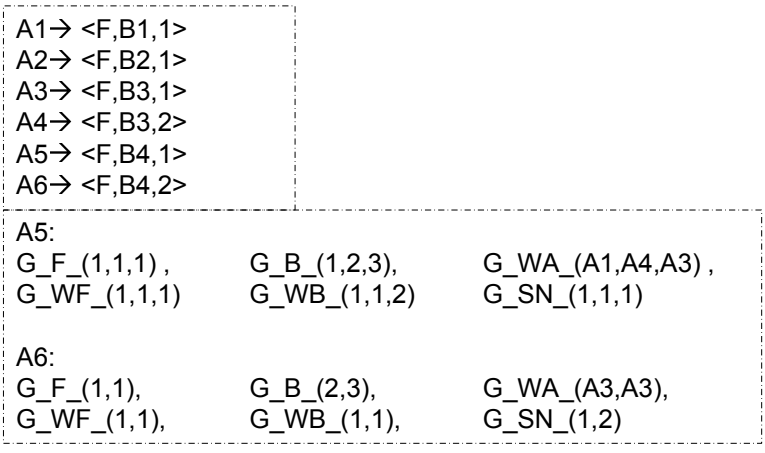


FIGURE 4.6. Compressing each subsequence using Sequitur.

Compressing the data flow sequence. After organizing the memory dependence edges according to their unique load instruction ids, all edges at the same point could be compacted using the Sequitur algorithm. As can be seen from Figure 4.5(b), the edges in each group share a significant similarity with each item individually. For this reason, the edges form 7 individual subsequences and get compressed separately. Moreover, the first subsequence is removed as all edges in the same group share the same global id. The last 6 subsequences are compressed and the results are shown in Figure 4.6.

4.2 Implementation and experiments

The TWPP+ algorithm has been implemented and evaluated to compress program traces for the programs from the SPECint95 benchmark suite [50] used in the preceding chapter. The original traces used in the experiments were generated using the Trimaran compiler infrastructure [55].

4.2.1 Compression results using TWPP+

The combined trace with both the control flow and memory address traces is usually much bigger than the control flow trace itself. Table 4.1 shows the control flow part and memory address part of the original trace. For 126.gcc and 132.ijpeg, the raw data is truncated at the upper limit of a file in our system, which is 2 gigabytes.

Program	Total control trace size (MB)	Total data trace size (MB)	Total trace size(MB)
`099.go`	176.0	555.9	731.9
`126.gcc`	164.4	687.1	851.1
`130.li`	84.9	295.7	380.6
`132.ijpeg`	130.0	838.0	968.0
`134.perl`	44.9	158.0	202.9

TABLE 4.1. Memory trace characteristics.

From the table, the length of the memory address trace is about four times bigger than that of the control flow trace. As discussed above, there are two reasons. First, a word (32 bits) is used to represent a memory address while a half word (16 bits) is used to represent a basic block id. There is a factor of 2 for this reason. Second, many basic blocks contain multiple memory accesses and thus each basic block instance maps to several recorded memory addresses.

The first conducted experiment was aimed at evaluating the effect of eliminating the explicit memory addresses and converting the trace to memory dependence edges. Since an edge is recorded at every load instruction but not a store instruction point, the effect of this conversion is determined by the percentage distribution of load and store instructions. Table 4.2 shows the static distribution of load and store instructions, and the dynamic distribution of load and store instances for different benchmarks.

The results indicate that 63.2% of all memory access instances will generate memory dependence edges at runtime. However, we know that each edge is indicated by two memory access points, so that without any compression, there are 2*63.2%=126.4% memory access points stored in the edge representation. Compared to the original memory address trace, the conversion introduces about 26% more access points.

Program	Static			Dynamic		
	load inst.	store inst.	load %	load inst.	store inst.	load %
`099.go`	11719	6682	63.7 %	100.5 M	38.5 M	72.3 %
`126.gcc`	34474	19649	63.7 %	108.2 M	63.5 M	63.0 %
`130.li`	1564	1396	52.8 %	45.1 M	28.8 M	61.0 %
`132.ijpeg`	3704	2840	56.6 %	141.8 M	67.9 M	67.6 %
`134.perl`	3548	2862	55.4 %	20.5 M	19.0 M	51.9 %
`Average`			58.4 %			63.2 %

TABLE 4.2. Distribution of load and store accesses.

The second step of the proposed compression algorithm removes the unnecessary

memory dependence edges and compact the edge representation. Edge compaction is done for every edge and an edge is packed into 6 items from its original 10 items, which means there is a 40% reduction. The dictionary generated is usually very small. For different benchmarks, their dictionary sizes are shown in Figure 4.3.

Program	Memory access point dictionary size (KB)
099.go	110.4
126.gcc	324.7
130.li	17.8
132.ijpeg	39.3
134.perl	38.5

TABLE 4.3. Dictionary size for memory access points.

Data was collected to estimate the percentage of edges that are removed and the actual percentage of edges removed from the sample traces. Table 4.4 summarizes the load (store) instructions with single and multiple source (destination) addresses. An edge, which consists of a load instance and a store instance, is removed only when its store instruction can only write to one address and its load instruction can only load from that address. Thus, the probability that we will drop a memory dependence edge is the product of the probability of a load is a single source load instruction and the probability of a store is a single destination store instruction. From the table, the estimation is that on average 24% of total data dependence edges would be removed.

Table 4.5 gives the actual percentage of edges removed from the sample traces. The total instances of load and stores in each category were collected and the actual edges removed from the sample traces. Dynamically, if a load instruction from the single source address category loads a value from an address which was written by a store instruction from the single destination address category, the corresponding edge is skipped. The results from the table show that about 17.7% of total memory dependence edges are removed. After removing these edges, about 53.3% of dynamic

Program	Load		Store		Estimation of removed edges
	multiple address	single address	multiple address	single address	
099.go	9048	2671	4434	2248	7.7 %
126.gcc	20406	14068	9495	10154	21.1%
130.li	1008	556	859	537	13.7%
132.ijpeg	1181	2523	836	2004	48.1%
134.perl	1617	1931	1256	1606	30.5%
Average					24.2%

TABLE 4.4. Distribution of static load and store points.

memory access points need to be traced. Since each edge has two points, now we need to trace about the same number of access points as that in the raw trace ($2 \times 53.3\% = 106.6\%$).

Program	Load		Store		% of edges removed %	% of dynamic points to trace (%)
	multiple src. instances(MB)	single src. instances(MB)	multiple dest. instances(MB)	single dest. instances(MB)		
099.go	92.6	7.9	33.4	5.1	7.7 %	66.7 %
126.gcc	91.1	17.1	54.4	9.1	13.4 %	54.6 %
130.li	36.9	8.2	25.6	3.2	16.4 %	51.0 %
132.ijpeg	111.8	30.0	41.6	26.3	17.0 %	56.1 %
134.perl	13.2	7.3	13.7	5.3	34.0 %	34.4 %
Average					17.7 %	53.3 %

TABLE 4.5. Dynamic behavior and removed edges.

After having the edges grouped at each load instruction point, the resulting subsequences are compacted. The results are shown in Table 4.6 with a comparison to the scheme that applies the Sequitur algorithm directly to the entire trace combined with control flow and memory address information. For 126.gcc and 132.ijpeg, both the uncompressed trace and the result contain only part of the control flow trace discussed in the preceding chapter.

Program	Uncompressed size (MB)	Sequitur size(MB)	TWPP+ size(MB)
099.go	731.9	195.2	317.6 + 26.5 = 344.1
126.gcc	851.1	114.0	146.1 + 13.8 = 159.9
130.li	380.6	35.9	24.4 + 6.4 = 30.8
132.ijpeg	968.0	45.8	55.2 + 6.0 = 61.2
134.perl	202.9	53.3	1.2 + 0.7 = 1.9

TABLE 4.6. Compression results using Sequitur and TWPP+.

4.2.2 Average scan length

The purpose of converting a memory address trace into a new TWPP+ representation is to help the construction of precise data flow information. Since the load and store instances of a memory dependence edge are usually separated from each other, we have to scan the two different representations in order to set up an edge. The next experiment evaluates the average length to be scanned in order to recover a memory dependence edge from different representations.

Given a load instance, the store instance of its dependence edge is found in the raw trace by backward traversal of the trace. In the worst case, the scan length can be as high as the length of the trace. The results in Figure 4.7 show the average length over all edges of all load points. In WPP representation, which is compressed by Sequitur algorithm, the worst-case scan length is up to the length of the compressed representation. In the WPP representation, the intermediate non-terminal symbols are merely grammar symbols and thus they cannot help in information retrieval. In many cases, the whole representation needs to be scanned to find the dependence. However, the results in Figure 4.7 is the best case estimation which means we need to just scan the compressed items between the load instance and store instance of the edge. In the new TWPP+ representation, the memory dependence edges are explicitly annotated to each load instruction. Once the load instance is found, we could find the store instance of the edge from searching all edges annotated to this

load. Since the edges are organized as several grammars, we count the length of these grammars and the results in Figure 4.7 show the average and maximal length that is scanned in the TWPP+ representation.

Program	Raw Trace	WPP	TWPP+	
			Average	Maximum
099.go	12.2 M	3.3 M	33.4 K	2.1 M
126.gcc	23.8 M	3.2 M	6.1 K	1.4 M
130.li	10.0 M	0.9 M	22.7 K	4.2 M
132.ijpeg	13.5 M	0.6 M	28.1 K	2.4 M
134.perl	7.8 M	2.0 M	0.5 K	0.7 M

TABLE 4.7. Average items scanned before finding a memory dependence edge.

From the table, we find it is orders of magnitude faster to find a data dependence edge using the TWPP+ representation than that using the WPP representation.

4.3 Related work

Chilimbi [15] proposed using the Sequitur [40] algorithm to compress memory address traces directly. The lossy compressed result, identified as whole program stream (WPS) is used to find hot subsequences of data object accesses and to use these subsequences to improve memory reference locality. WPS is not a suitable representation for memory dependence analysis because of the following reasons. First, the address abstraction is used before compression. The abstracted data reference trace consists of units of larger granularity, e.g. object ids instead of field ids. Thus the precise memory dependence information at word level is lost. Second, the algorithm iterates several times and infrequently used memory addresses are discarded after each iteration. As a result, the precise memory dependence information is lost even at the large granularity. In contrast, the TWPP+ representation puts more emphasis on analysis and keeps the precise memory dependence information. Moreover, as discussed, ex-

plicit memory addresses are discarded as the goal of TWPP+ is not to improve the data locality.

4.4 Conclusion

To assist in data flow analysis with precise data and control flow information for a given execution, a complete whole program path with both control flow profiles and memory addresses is collected. However, it is observed that the explicit memory addresses are not necessary for many applications. Following the same design philosophy as the one used in designing TWPP, a new representation was proposed in this chapter to reorganize the memory address trace into a sequence of memory dependence edges annotated on each load instruction point. While providing more precise information during data flow analysis, the new representation achieves compression results comparable to that using Sequitur directly on the combined trace. Moreover, the estimated speed to determine a memory dependence edge from the TWPP+ representation is orders of magnitude faster than that from the WPP representation.

Chapter 5
Applications of TWPP+

The preceding two chapters introduced the new timestamped whole program path (TWPP+) representation to compress a complete program trace which contains both data flow and memory dependence information. The new representation has the advantage of helping the program analyses and optimizations in several ways. This chapter introduces three different types of applications using TWPP+.

In the TWPP representation, all the execution information related to a specific program entity is organized together. For example, each basic block groups its function level execution information as a sequence of timestamps; a load instruction groups all of its memory dependence edges and compresses them together. In such a representation, *summary information* with respect to different entities can be easily collected. By counting the number of items annotated to each entity, simple frequency information could be collected and used to find hot program regions. Section 5.1 discusses a more complex application which uses TWPP+ to decide the percentage distribution of redundant load and store instructions.

Although the execution information about different program entities has been separated, TWPP keeps the global timestamps so that the original execution order could be reconstructed easily. This order is especially helpful in finding whether a given data flow fact holds at a given program point and with what frequency. Section 5.2 discusses how to collect such information in a demand-driven fashion.

The complete control flow and memory dependence information could also be used in other applications. In section 5.3, TWPP is used in implementing different dynamic slicing algorithms with trade off between cost of computing slices and their accuracy.

5.1 Exploring LOAD/STORE redundancy

In modern architectures, memory accesses have a long latency and thus a significant amount of research has been carried out to reduce the number of load and store instructions. However, before a load or store instruction can be removed, it must be identified as being redundant. This section will show how to assist in this type of optimization with memory dependence edges recorded in TWPP.

5.1.1 Identifying a redundant LOAD/STORE instruction

A load instruction of the form "`LD R1, off(R2)`" loads the value from the memory address (R2+off) into register R1. A store instruction of the form "`ST R1, off(R2)`" stores the value from register R1 into the memory address (R2+off).

Redundant LOAD and STORE instructions are defined as follows. A load instruction instance l which loads from a memory address m is identified as a redundant load if it satisfies the following conditions.

- There is another load instruction instance l' which is executed before l, and l' loads from the same memory address m. l and l' could be the instances of the same instruction or two different instructions.
- There is no store instruction instance s between l' and l such that s writes to the memory address m.

A store instruction instance s which writes to a memory address m is identified as a redundant store if it satisfies the following conditions.

- There is another store instruction instance s' which is executed after s, and s' stores to the same memory address m. s and s' could be the instances of the same instruction or two different instructions.

- There is no load instruction instance l between s and s' such that l loads from the memory address m.

For example, in Figure 5.1, the load instance L is redundant since load instance L2 gets the value from the same address and the address is not overwritten in between. The store instance S is redundant since there is no load that gets its value before it is overwritten.

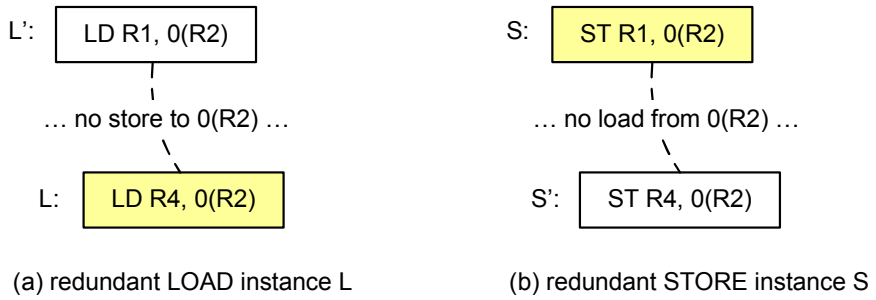


FIGURE 5.1. Redundant LOAD/STORE instructions.

Without program profiles, a compiler could remove a redundant load (or store) instruction only when it is ensured that all of its instances are redundant. However, in many cases, the memory addresses of different load and store instructions are statically determined as potentially aliased but dynamically never overlap. Even with the most advanced aliasing analysis, many fully redundant load and store instructions cannot be identified as redundant and be removed. In addition, it is desirable to exploit those instructions each of which has many of its instances as redundant. These opportunities could be discovered with the help of memory dependence edges stored in the TWPP+ representation. Next, we study the potential redundant load and store instances and their distribution. As an ideal study, the following two assumptions are made.

- There are unlimited number of registers to hold values loaded from the memory.

- Given a memory address, if its latest copy has been loaded to a register, that register can always be identified.

5.1.2 Identifying redundant loads from TWPP+

Since there are no explicit memory addresses in a TWPP+ representation, load redundancy is not identified by comparing different load addresses, but instead it is detected through explicit memory dependence edges.

Handle the discarded memory dependence edges. A TWPP+ representation discards those edges that can be reconstructed from the control flow trace. From the preceding chapter, we know these edges are about 18% of total edges. The discarded edges share one common property, i.e. the load and store instances of these edges are instances of load and store instructions each can load from or store to only *one* statically known address.

There are three possible ways to handle these discarded edges. First, we consider only the recorded edges and their corresponding instructions. Because of the property discussed above – single memory address that is statically known, the load and store instructions involved in discarded edges are relatively easy cases to handle. Thus we can skip processing them. Second, if we can integrate them into the TWPP+ representation, as we show in the preceding chapter, it increases about 18% uncompressed edges. Third, we can recover them from a control flow graph traversal. However, because we are only interested in recovering these discarded edges, the control flow graph is significantly simplified.

Here, we give the algorithm to recover these edges before finding load and store redundancy. First, instructions are selected from the set of all load and store instructions. Each selected instruction can load from or store to only *one* memory address and the address is statically decidable. Second, the control flow graph of a function is simplified by removing all basic blocks if they do not contain any selected instruc-

tions or any function calls. A function is discarded if after the above simplification, it contains nothing. The above process is applied iteratively until the control flow graph does not change any more. Third, the dynamic call graph, as well as a function's control flow graph, are traversed backwards. During the traversal, a memory dependence edge is set up between a load instance and its immediate preceding store instance which accesses the same memory address.

Identify redundant loads. As described in the preceding chapter, there is a memory dependence edge for each load instance and the edge is of the form $[< G_1, FI_1, BI_1 >, < G_2, FI_2, BI_2 >, SS]$ where G,FI,BI denote the global id, function instance id and basic block instance id and SS denotes this load is the SS-th load of the store value. The conditions to identify a redundant load can now be restated as follows.

A load instance l denoted by an edge $[< G_1, FI_1, BI_1 >, < G_2, FI_2, BI_2 >, SS_1]$ is redundant if there is another load l' denoted by another edge $[< G_3, FI_3, BI_3 >, < G_4, FI_4, BI_4 >, SS_2]$ and

- l and l' load the value from the same memory address and there is no store instruction in between that writes to this memory address. This means they get the value from the same store instance, i.e. $< G_2, FI_2, BI_2 >=< G_4, FI_4, BI_4 >$.
- l is executed before l'. This means $< G_1, FI_1, BI_1 >$ has a smaller timestamp than $< G_3, FI_3, BI_3 >$, i.e. $SS_1 < SS_2$.

For example, in Figure 5.2(a), the store instruction S1 has been executed 3 times. For its second instance, there are two load instructions L1 and L2 and the second instance of each load instruction gets the value from it. If the second instance of L1 is executed after that of L2, then it is redundant, otherwise the second instance of L2 is redundant.

We know that memory dependence edges are organized at individual load points and stored in separated blocks, shown in Figure 5.2(b). According to the conditions

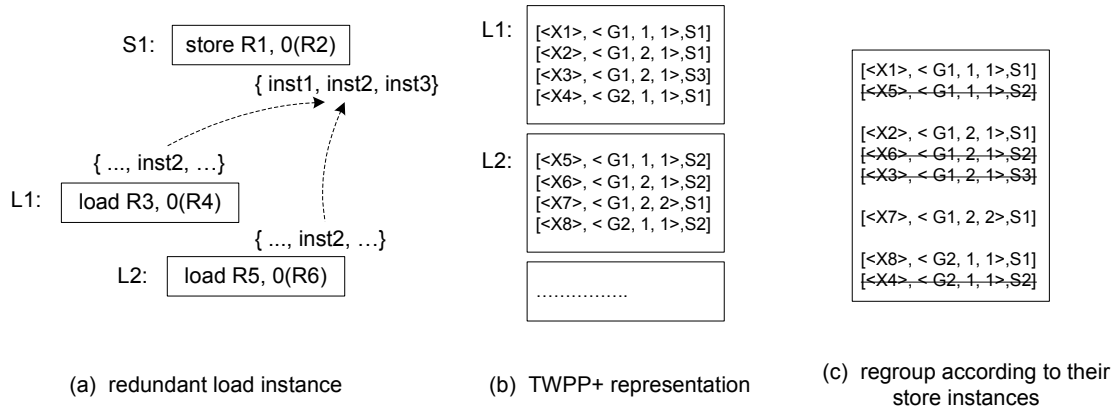


FIGURE 5.2. Determining a redundant load.

discussed above, we can identify redundant loads from TWPP+ representation as follows. The first step is to recover the dependence edges discarded from the TWPP+ and group them as a block. The second step is combine all edges and regroup them according to their store instances. For the same store instance, edges are sorted according to their load timestamps. Finally, we mark all loads except the first load in each group as redundant load instances and summarize the information (Figure 5.2(c)).

5.1.3 Identifying redundant stores from TWPP+

We can identify redundant stores from TWPP+ similarly. According to the conditions that identify a redundant store, there is no load instance which gets a value from a redundant store instance. Thus if an instance is involved in any edge, it is not redundant. Otherwise it is redundant.

For example, in Figure 5.3(a), the store instruction S1 has been executed 3 times, both the first and the third instances have their dependent load instruction instances. There is no load instruction which gets the value from its second instance. The second instance is a redundant store instance. If the store is writing to an output stream, it is never marked as being redundant.

Identifying redundant stores from the TWPP+ representation is similar to identifying redundant loads. First, discarded edges are recovered. Second, all edges are grouped and sorted. From the sorted list, all skipped instance number of a store instruction denotes a redundant instance, shown in Figure 5.3(c). [1]

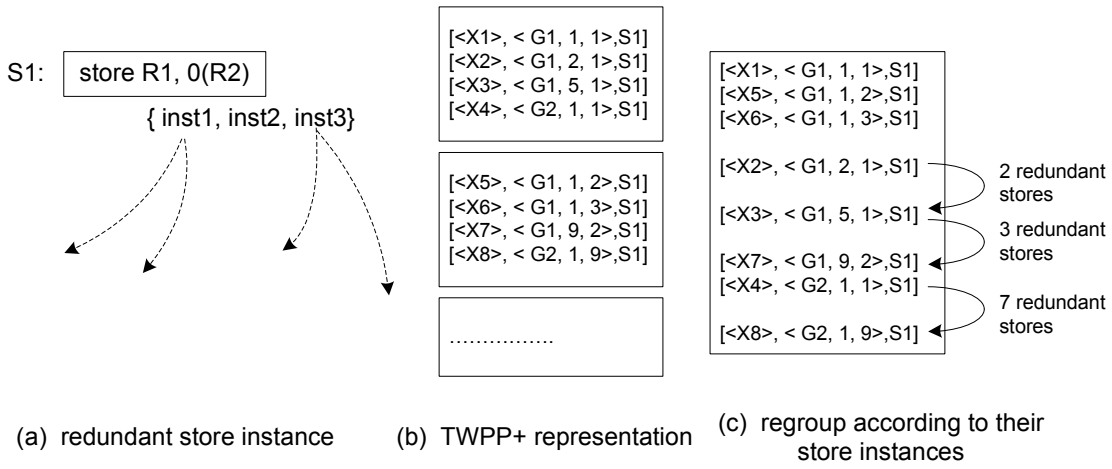


FIGURE 5.3. Determining all redundant stores from TWPP+.

5.1.4 Experimental results

Redundant load and store instructions for SPECint95 benchmark programs are evaluated using the algorithm described above. Each load and store instruction is uniquely numbered and individually analyzed.

First we study the distribution of redundant LOAD instances. It is to find for each given instruction, the percentage of its instances that are redundant. For example, if 90 out of 100 instances for a load instruction are redundant, it is categorized as a 90% redundant instruction. The first bar in Figure 5.4 shows the distribution of load instructions for different programs.

Although an instruction has a high percentage of redundancy, it might have only a small number of instances and thus not be very *important*. We weighted all instruc-

[1] To assist analysis, the highest instance number of each store instruction is recorded during profiling.

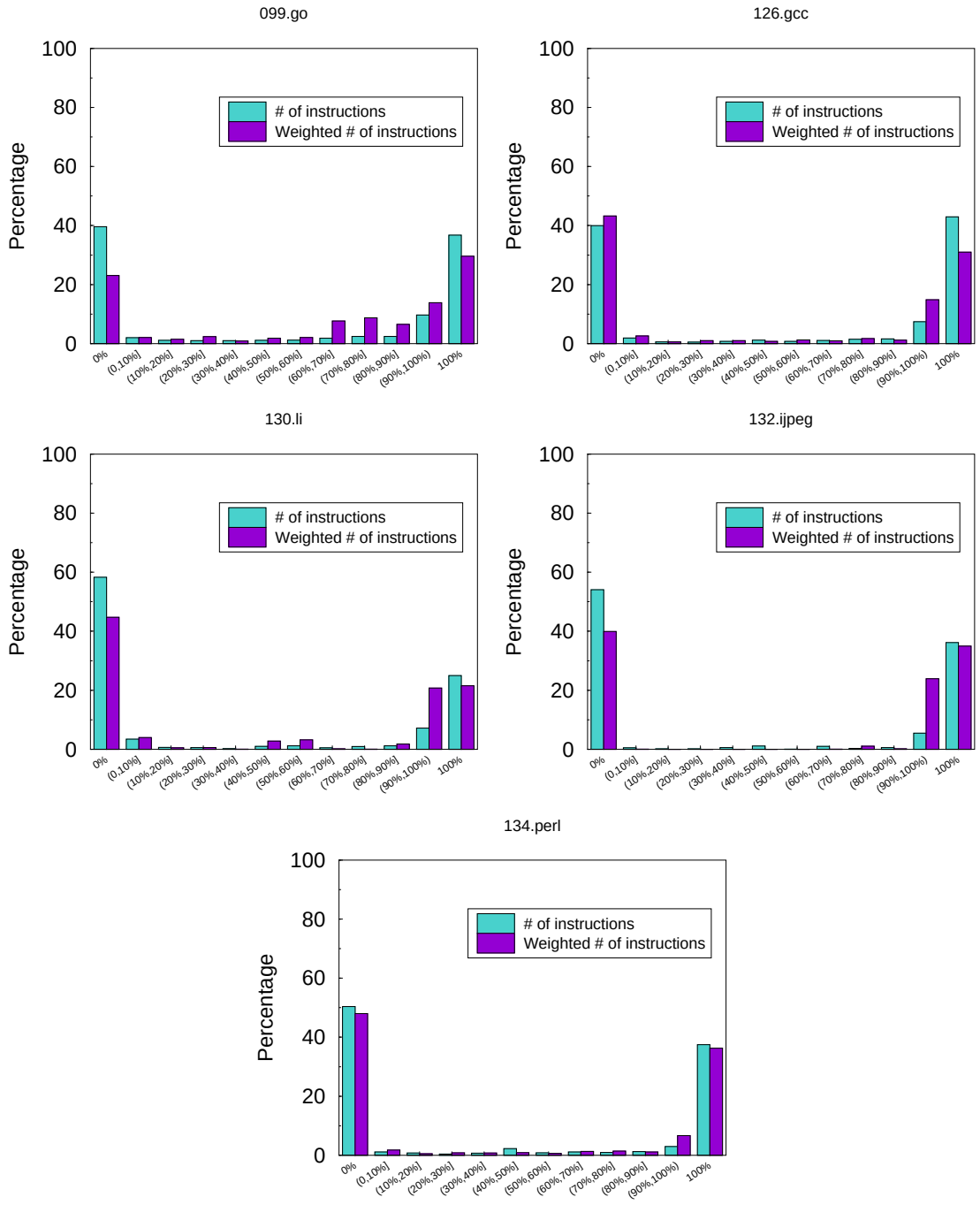


FIGURE 5.4. Ideal LOAD redundancy

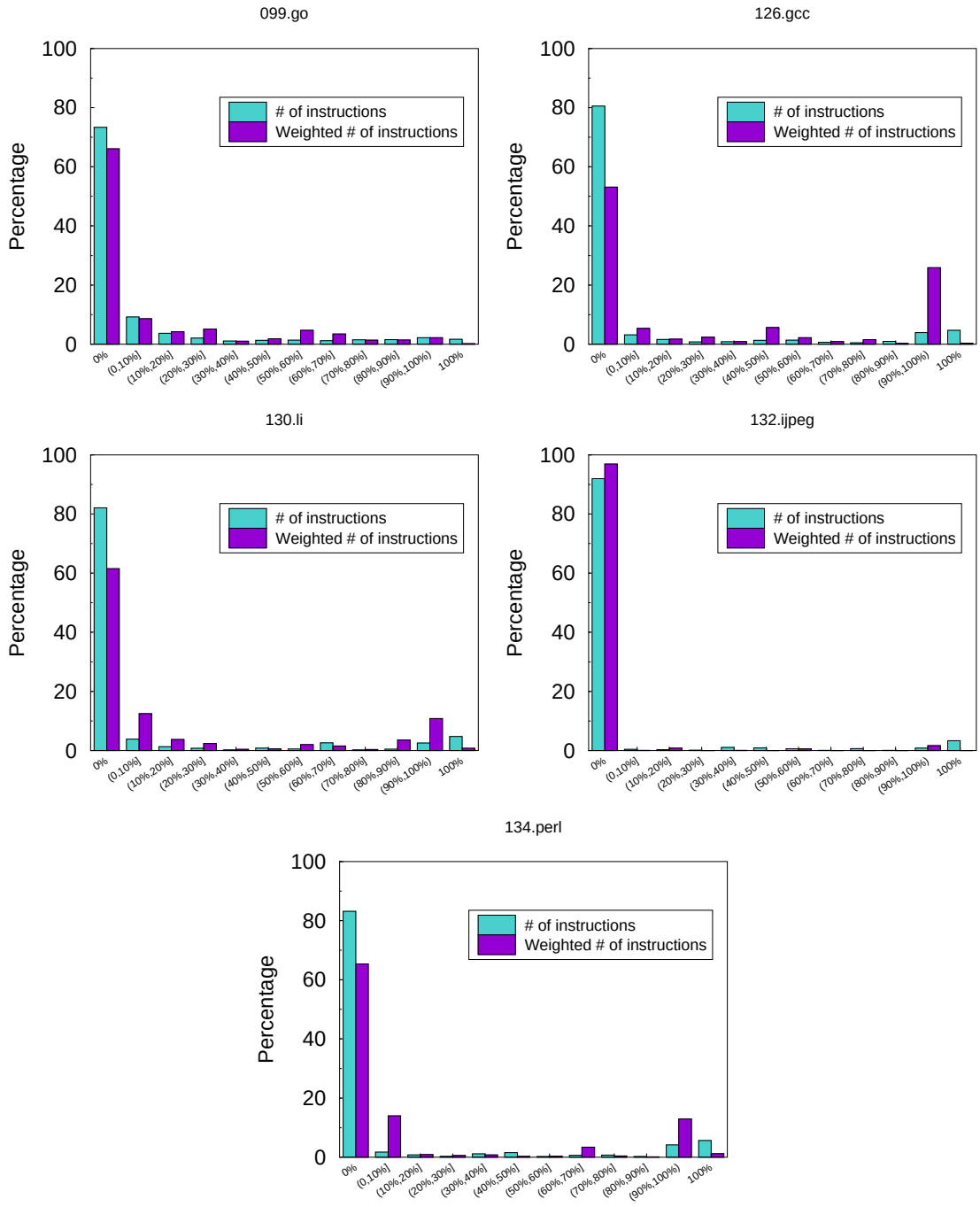


FIGURE 5.5. Ideal STORE redundancy

tions by summarizing all instances in each category and the results are shown by the second bar in Figure 5.4.

From the figure, most load instructions fall in to three categories: "0%", "90-100%" or "100%". If a load belongs to the "0%" category, there is no opportunity to optimize it; however, it is good to separate them from the rest because we do not have to invest any compile time on further analyzing them. If a load belongs to the "90-100%" or "100%" category, it is worthy of further analysis since good benefits are expected from optimizing it.

The experimental results for STORE redundancy (Figure 5.5) show that most store instructions are not redundant at all. There are not many opportunities for optimizing store instructions.

5.2 Frequency of data flow facts

Although TWPP+ reorganizes the complete program trace, it keeps the timestamp information such that the original execution order could be reconstructed. The exact execution order is very important in determining if some data flow facts hold and their frequencies at some points during the execution. During profile guided compile-time optimization and dynamic optimization of programs, one example could be a query which looks like: *How often does a data flow fact hold at a program point during the execution captured by the WPP?*. This query is useful for identifying *hot data flow facts*, i.e., data flow facts that hold very often during the execution. Another useful query is: *Does a data flow fact hold at a given program point during the execution captured by the WPP?*. This query is useful during debugging of programs including during dynamic slice computation which is discussed in the next section.

In this section, a *profile-limited data flow analysis* approach is introduced to collect information about data flow facts with respect to a given whole program path (WPP). The analysis presented in this section can be used to answer the above types of queries.

For this analysis, there is no need to access the entire TWPP but only a subset of information corresponding to the function under consideration. In particular, a timestamp annotated dynamic control flow graph is used for the given path trace which is described below.

5.2.1 Timestamp annotated dynamic CFG

This representation consists of the dynamic control flow graph in which DBBs are annotated with timestamp vectors. This representation is quite adequate for data flow analysis because we can trace the WPP using the timestamp vectors associated with the dynamic basic blocks and limit the exploration of only those control subpaths that appear as part of the WPP during data flow analysis. The following characteristics make this proposed representation particularly attractive for profile-limited data flow analysis.

First it allows efficient backward and forward traversal of the path trace starting from any arbitrary point in the path trace. A timestamp and program point pair (t, n) together specify a particular point in the path trace. The preceding point is $(t-1, m)$ where m is the predecessor of n in the dynamic control flow graph labeled with timestamp $t-1$. Similarly the succeeding point is $(t+1, s)$ where s is a successor of n in the dynamic control flow graph which is labeled with timestamp $t+1$.

Second it allows efficient simultaneous traversals of multiple subpaths in the path trace. A vector of timestamps at a program point $(\vec{\mathcal{T}}, n)$ can be used to represent multiple traversal points. Each element in the vector can be incremented or decremented and resulting timestamps can be matched with timestamps of predecessors and successors to continue simultaneous traversal along multiple subpaths. *Compaction of timestamps directly attributes to the efficiency of traversals.* For example consider a series of timestamps represented by (2:20:2) in our representation. A simple increment/decrement resulting in (3:21:2)/(1:19:2) corresponds to simultaneous

forward/backward traversal along 10 subpaths in the path trace.

An indicator of the relative costs of profile-limited analysis and traditional static analysis are the cumulative sizes of static and dynamic flow graphs (see Table 5.1). We compared the total number of nodes (N) and edges (E) in the static and dynamic flow graphs. For a given function multiple dynamic flow graphs can result because of multiple unique traces associated with it. The nodes and edges in all of these graphs were counted in computing the cumulative size of the dynamic flow graphs. From the results in Table 5.1 we can see that the number of nodes and edges in the dynamic graphs are typically much smaller than those in the static graphs. However, the cost of profile-limited analysis is also dependent upon the size of timestamp vector associated with each node. Average size of the timestamp vector is shown in the last column of Table 5.1 (the value in parenthesis is the size of the vector before compaction - the results show that timestamp vector is significantly reduced in size using our compaction technique). In summary, the data in Table 5.1 indicates that while, as expected, profile-limited analysis is more expensive than static analysis, it has a reasonable cost.

Program	Static FG		Dynamic FG		
	N	E	$\sum$ N	$\sum$ E	avg. $\lvert\vec{\mathcal{T}}\rvert$
`126.gcc`	66571	104379	8838	20012	14.0 (33.1)
`132.ijpeg`	5718	8105	754	1213	18.1 (109.7)
`099.go`	10823	16236	4739	16591	11.9 (15.7)
`130.li`	2701	3536	265	289	51.2 (410.3)
`134.perl`	13117	19539	501	674	3.9 (616.8)

TABLE 5.1. Sizes of static and dynamic flow graphs.

5.2.2 Demand-driven analysis

A traditional data flow analysis framework *created* (GEN) and *removed* (KILL) data flow facts at each basic block. The *transfer* function is used to propagate the data

flow facts through a basic block and compute the data flow solutions. Generally, conservative solutions are computed at the *meet* or *split* points as well as for loops whose solutions are computed iteratively. Details about data flow analysis can be found in [38].

It is natural to formulate profile-limited analysis in a demand-driven fashion [20, 46]. This is because the applications of profile-limited analysis request information incrementally. For example, during debugging a user typically makes a request for the dynamic slice corresponding to only one variable at a fixed program point (i.e., we only need to compute subset of data flow information for subset of program points). Similarly during profile-guided or dynamic code optimization, subset of profile-limited data flow information may be requested by the optimizer for subset of program points in hot regions of the program [8].

Queries for profile-limited data flow

A *profile-limited data flow query* is of the form $< \mathcal{T}, n >_d$, where n is a node, $\mathcal{T}$ is a subset of timestamps for n in the path trace, i.e., $\mathcal{T} \subseteq \mathcal{T}(n)$, and d is the data flow fact of interest. This query represents a request for determining whether or not d holds true prior to n's executions corresponding to timestamp values in $\mathcal{T}$. Therefore the query $< \mathcal{T}(n), n >_d$ determines the data flow solution corresponding to all executions of n in a given path trace. The solution to this query allows us to determine if d always holds true, never holds true, and holds true sometimes for the given path trace. In fact solving such queries allows us to determine the frequency with which d holds true with respect to the given path trace [45, 7, 22, 24, 23].

Query propagation

Let us consider profile-limited demand-driven *backward* propagation of queries for GEN-KILL problems because they arise both during code optimization and debug-

ging. For simplicity, the analysis of only intraprocedural paths is considered. However, in analyzing these paths the effects of any function calls that a path trace may contain will be taken into account. The technique presented can be easily extended to handle interprocedural paths by analyzing path traces of multiple functions in concert and propagating queries along interprocedural paths [20].

The demand-driven propagation begins at a point n when the query $< \mathcal{T}, n >_d$ is raised. For GEN-KILL problems it is appropriate to propagate a *timestamp vector*, $\vec{\mathcal{T}}$, which contains one slot for every timestamp, or more precisely, for every entry in the compacted TWPP path trace. The propagation should be viewed as simultaneous (or parallel) search for data flow solutions corresponding to each timestamp in $\mathcal{T}$. Each slot in $\vec{\mathcal{T}}$ is initialized to the timestamp value(s) to which it corresponds. The propagation of this $\vec{\mathcal{T}}$ begins at n.

It must be ensured that query propagation is consistent with the path trace under consideration. As discussed earlier in this section, this goal is easily accomplished using the timestamp annotated dynamic control flow graph representation. It is possible to correctly manipulate the timestamp vector during propagation such that the timestamps in the vector are propagated only to the appropriate predecessors. When a node that answers the query (true or false) with respect to a particular timestamp is encountered, the propagation on behalf of that timestamp ceases. Otherwise equivalent queries are generated and propagated along the path trace.

The query $< \vec{\mathcal{T}}, n >$ represents the search for dynamic GEN-KILL points corresponding to timestamps of n for which slots were created in $\vec{\mathcal{T}}$. For carrying out the propagation first dynamic GEN-KILL sets (i.e, sets w.r.t. to a given TWPP) for a data flow fact d (which are denoted as $DGEN_n^d$ and $DKILL_n^d$) must be computed. Although n is a dynamic basic block, to simplify the presentation it is assumed that n contains a single statement. If node n contains a call to function f, then the traces for calls made by the n's instances corresponding to $\mathcal{T}(n)$ are examined. The set $GEN_f^d(\mathcal{T}(n))$ ($KILL_f^d(\mathcal{T}(n))$) contains the subset of timestamps from $\mathcal{T}(n)$ for

which call to function f generates (kills) d. If node n simply contains a statement, the dynamic sets are computed from the static GEN and KILL sets for node n denoted below as $SGEN_n$ and $SKILL_n$.

$$DGEN_n^d = \begin{cases} GEN_f^d(\mathcal{T}(n)) & if\ n\ calls\ f \\ \mathcal{T}(n) & elseif\ d \in SGEN_n \\ \phi & otherwise \end{cases}$$

$$DKILL_n^d = \begin{cases} KILL_f^d(\mathcal{T}(n)) & if\ n\ calls\ f \\ \mathcal{T}(n) & elseif\ d \in SKILL_n \\ \phi & otherwise \end{cases}$$

Now let us consider query propagation. The timestamp values in $\vec{\mathcal{T}}$ are each decremented by 1 during every step of backward propagation. Only those resulting timestamp values which are present in $\mathcal{T}(m)$, where m is a predecessor node, are propagated to m. At m the query for a timestamp may be resolved as true (if $t \in DGEN_m^d$) or as false (if $t \in DKILL_m^d$). If it is not resolved, then the above process is repeated starting with the decrementing of the timestamp and propagation continues. It should be noted that only a subset of slots may be relevant for a given predecessor node; thus the other slots will contain a null value denoted by $\perp$. The above rules are stated precisely below and are further illustrated by example applications discussed in the subsequent sections.

Propagation of $< \vec{\mathcal{T}}, n >$
Notation: $\vec{\mathcal{T}}/\mathcal{T}'$ is a timestamp vector st $(\vec{\mathcal{T}}/\mathcal{T}')_i$ = if $(\vec{\mathcal{T}})_i \in \mathcal{T}'$ then $(\vec{\mathcal{T}})_i$ else $\perp$.
Slots in $\vec{\mathcal{T}}$ resolved as true are slots in vectors $\bigcup_{m \in pred(n)} (\vec{\mathcal{T}} - \vec{1})\ /\ DGEN_m^d$ which do not contain $\perp$. Slots in $\vec{\mathcal{T}}$ resolved as false are slots in vectors $\bigcup_{m \in pred(n)} (\vec{\mathcal{T}} - \vec{1})\ /\ DKILL_m^d$ which do not contain $\perp$. Queries propagated for unresolved slots in $\vec{\mathcal{T}}$ $\bigcup_{m \in pred(n)} < (\vec{\mathcal{T}} - \vec{1})\ /\ (\mathcal{T}(m) - DGEN_m^d - DKILL_m^d), m >$

5.3 Dynamic program slicing with TWPP

Program slicing is a useful tool in program analysis, understanding and debugging. Given a program point P of a program S and a variable V, a *static slice* computes the set of statements whose execution could possibly affect the value of V in some executions. Given an execution history, a program point P of a program S and a variable V, a *dynamic slice* computes the set of statements whose execution affect the value of V in this execution history. For example, in Figure 5.6, we have

$$\begin{aligned} \textit{static slice}(Z,(14)) &= \{1,2,3,4,5,6,7,8,9,10,11,13,14,15,16\}; \\ \textit{dynamic slice}(Z,(14),\textit{trace}) &= \{2,3,4,6,7,8,9,11,13,14,15,16\}. \end{aligned}$$

There are two kinds of dependences: data dependences and control dependences. For example, in Figure 5.6, variable V is control dependent on statements 6,8 and data dependent on the rest.

Static backward program slicing was first proposed by Weiser as a debugging aid [59]. The more precise dynamic slicing was proposed by Korel and Laski [30]. Most recently Agrawal and Horgan [1] developed three dynamic slicing algorithms which trade-off precision in the computed slice with the time it takes to compute the slice. Each of these algorithms constructs a *different* specialized program dependence graph (PDG) to capture the dependences exercised in a given execution. A backward traversal over the graph is used to compute the dynamic slice as a transitive closure over data and control dependences. Each of the above dynamic slicing algorithms can be implemented using one common representation, the timestamped dynamic control flow graph, and thus the construction of specialized graphs suggested in [1] is avoided.

5.3.1 Precise dynamic slicing with TWPP+

Three algorithms with different accuracy and costs are presented in [1] to calculate dynamic slices. The implementation of the precise algorithm with TWPP+ is dis-

cussed below. The imprecise algorithms 1 and 2, which compromise the accuracy for speed, will be discussed later.

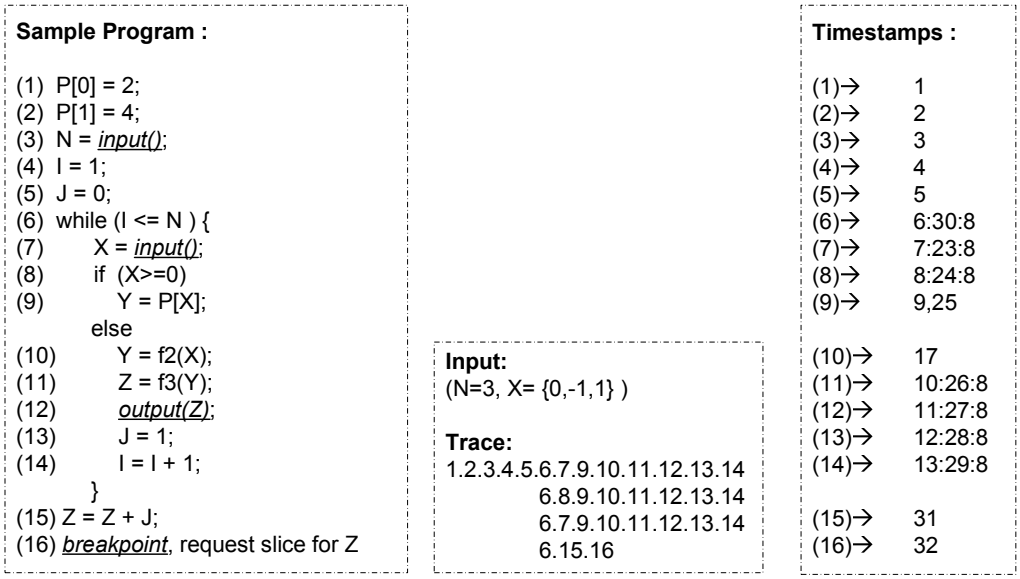


FIGURE 5.6. Dynamic slicing example.

Precise algorithm: This method duplicates the executed node and its dependence edges during the execution so that it can distinguish between the instances of a given statement. The expanded PDG graph is traversed to find the precise dynamic slice. The backward analysis uses timestamps to find dependences and when a dependence is found only a single timestamp is added to the newly generated queries. In other words we identify the precise instance of the assignment (for data dependence) and predicate (for control dependence) which is the source of the dependence and generate queries only for the corresponding instances of variables that are read by the assignment or predicate. In our example, note that although statements 10 and 5 are executed, they are not included in the slice because the value of Z at 15 depends only upon the values of Y and J computed by statements 9 and 13.

The implementation of algorithm 3 using TWPP+ is shown in Figure 5.7. It traverses the TWPP+ representation backwards and includes both control and data

Dynamic Slicing Algorithm:

```
(01)  Q = {< TS_0, S_0 >_X};
(02)  WHILE ( Q != NULL)
(03)     item = dequeue (Q);
(04)     IF ( item is "< TS_1, S_1 >" )
(05)        IF ( S_1 is a control statement )
(06)           insert (S_1, TS_1) to Dslice;
(07)           FOR each variable Z in S_1
(08)              insert < TS_1, S_1 >_Z into Q;
(09)        ELSE
(10)           FOR each preceding statement S_2 of S_1
(11)              insert < TS_1 - 1, S_2 > into Q;
(12)     IF ( item is "< TS_1, S_1 >_Y" )
(13)        IF ( S_1 is a control statement )
(14)           insert (S_1, TS_1) to Dslice;
(15)           FOR each variable Z in S_1
(16)              insert < TS_1, S_1 >_Z into Q;
(17)        IF ( S_1 writes to Y )
(18)           insert (S_1, TS_1) to Dslice;
(19)           FOR each RHS variable Z of S_1
(20)           IF ( data dependence edge list contains Z with TS_1)
(21)              edge := (..., < G_2, I_2 >)
(22)              insert < G_2, I_2 > into Q;
(23)           ELSE
(24)              FOR each preceding statement S_2 of S_1
(25)                 insert < TS_1 - 1, S_2 > into Q;
(26)        IF ( S_1 writes to some other variable )
(27)           FOR each preceding statement S_2 of S_1
(28)              insert < TS_1 - 1, S_2 >_Y into Q;
(29)     IF ( item is "< G_1, I_1 >" )
(30)        find the timestamp TS_2 for instance I_1 at global load point G_1
(31)        find the statement S_2 for global load point G_1
(32)        insert (S_2, TS) to Dslice;
(33)        FOR each RHS variable Z of S_2
(34)           insert < TS_2, S_2 >_Z into Q;
```

FIGURE 5.7. Precise dynamic slicing algorithm with TWPP+.

dependent statements into the dynamic slice. Control dependent statements are included from the backwards traversal of the control flow trace. However, there are two kinds of data dependent statements. If the load instance has a memory dependence edge, the algorithm follows this edge directly to its definition point and include the definition instance into the dynamic slice. Otherwise, it traverses along the control flow graph with the timestamp until the definition point is found.

The main data structure in this algorithm is a global queue which holds the items needed to be checked for further dependence. The queue could contain a mixed of three types of nodes and the algorithm is to handle them accordingly.

- $< TS, S >_X$. It means the algorithm needs to find both the data and control dependent edges for a access point of variable X. If S is a control block, the statement is found which the access of variable is control dependent on. If S writes to X, the definition of X is found, the algorithm can then start to resolve the dependence of its RHS variables. If S writes to a memory location other than X, the algorithm will skip this statement and go backwards further.

- $< TS, S >$. It means the data dependency has been resolved. The algorithm just need to go backwards and find the control dependency.

- $< G, I >$. It denotes that a recorded data dependence edge has been found and $< G, I >$ is the definition point of this edge. The instruction should be included into the sliced and data and control dependence should go further from that point.

The detailed propagation of queries for this algorithms is shown in Figure 5.8. The queries of the form $< \mathcal{T}, n >_V$ where $\mathcal{T}$ is the timestamp vector, n is the node at which the query is to be evaluated, and V is the variable whose definition is to be found. Therefore, a request for a slice on Z at line 16 is translated into the query $< [32], 16 >_Z$. The updated slice after the processing of a query is given in the

corresponding entry of the second column and the type of dependence (control or data) that caused the addition of a statement to the slice is also indicated.

Approach 3: Slicing request: $< [32], 14 >_Z$		
Query	Slice	Dependence
$< [32], 16 >_Z$	{16}	
$< [31], 15 >_Z < [31], 15 >_J$	{15,16}	data
$< [30], 6 >_Z < [30], 15 >_J < [30], 6 >_I < [30], 6 >_N$	{6,15,16}	control
$< [29], 14 >_Z < [29], 14 >_J < [29], 14 >_I < [29], 14 >_N$	{6,14,15,16}	data
$< [28], 13 >_Z < [28], 13 >_I < [28], 13 >_N$	{6,13,14,15,16}	data
$< [27], 12 >_Z < [27], 12 >_I < [27], 12 >_N$	{6,13,14,15,16}	data
$< [26], 11 >_Y < [26], 11 >_I < [26], 11 >_N$	{6,11,13,14,15,16}	data
$< [25], 9 >_X < [25], 9 >_I < [25], 9 >_N < [25], 9 >_{P[1]}$	{6,9,11,13,14,15,16}	data
$< [25], 9 >_X < [25], 9 >_I < [25], 9 >_N$	{2,6,9,11,13,14,15,16}	memory
$< [24], 8 >_X < [24], 8 >_I < [24], 8 >_N$	{2,6,8,9,11,13,14,15,16}	control
...	...	
$< [5], 5 >_I < [5], 5 >_N$	{2,4,6,7,8,9,11,13,14,15,16}	data
$< [4], 4 >_N$	{2,3,4,6,7,8,9,11,13,14,15,16}	data

FIGURE 5.8. Implementing A&H's dynamic slicing algorithm 3.

The worst case time complexity of the implementation using TWPP is the same as that of Agrawal and Horgan's algorithm. The primary cost of both algorithms comes from processing the control flow trace. The new algorithm must examine the entire trace to compute the TWPP path trace representation while their algorithm must examine the trace to construct a dynamic dependence graph. The main difference between the two approaches is as follows. Agrawal *et al.* compute all dynamic dependences first and construct a graph using which any dynamic slice request can be processed using a simple traversal. In contrast TWPP+ based approach computes relevant dependences for slicing requests upon demand (like Weiser's algorithm [59]). Since the same dependences may be relevant to different slicing requests, their recomputation must be avoided by caching the computed dependences. In other words TWPP+ based approach builds the dynamic dependence graph incrementally as slicing requests are processed.

Approach 1:Slicing request: $< *, 16 >_Z$		
Query	Slice	Dependence
$< *, 16 >_Z$	{16}	
$< *, 15 >_Z < *, 15 >_J$	{15,16}	data
$< *, 6 >_Z < *, 6 >_J < *, 6 >_I < *, 6 >_N$	{6,15,16}	control
$< *, 5 >_Z < *, 5 >_I < *, 5 >_N < *, 14 >_Z < *, 14 >_J < *, 14 >_I < *, 14 >_N$	{5,6,13,14,16}	data
$< *, 4 >_Z < *, 4 >_N < *, 13 >_Z < *, 13 >_I < *, 13 >_N$	{4,5,6,13,14,15,16}	data
$< *, 3 >_Z < *, 12 >_Z < *, 12 >_I < *, 12 >_N$	{3,4,5,6,13,14,15,16}	data
$< *, 2 >_Z < *, 11 >_Y < *, 11 >_I < *, 11 >_N$	{3,4,5,6,11,13,14,15,16}	data
$< *, 1 >_Z < *, 10 >_X < *, 10 >_I < *, 10 >_N < *, 9 >_{P[0],P[1]} < *, 9 >_X < *, 9 >_I < *, 9 >_N$	{3,4,5,6,9,10,11,13,14,15,16}	data
$< *, 10 >_X < *, 10 >_I < *, 10 >_N < *, 9 >_X < *, 9 >_I < *, 9 >_N$	{1,2,3,4,5,6,9,10,11,13,14,15,16}	memory
$< *, 8 >_X < *, 8 >_I < *, 8 >_N$	{1,2,3,4,5,6,8,9,10,11,13,14,15,16}	data,control
$< *, 5 >_I < *, 5 >_N$	{1,2,3,4,5,6,7,8,9,10,11,13,14,15,16}	data
$< *, 4 >_I < *, 4 >_N$	{1,2,3,4,5,6,7,8,9,10,11,13,14,15,16}	solved queries
Approach 2: Slicing request: $< [32], 16 >_Z$		
Query	Slice	Dependence
$< [32], 16 >_Z$	{16}	
$< [31], 15 >_Z < [31], 15 >_J$	{15,16}	data
$< [30], 6 >_Z < [30], 6 >_J < [6 : 30 : 8], 6 >_I < [6 : 30 : 8], 6 >_N$	{6,15,16}	control
$< [29], 14 >_Z < [29], 14 >_J < [5], 5 >_I < [13 : 29 : 8], 14 >_I < [5], 5 >_N < [13 : 29 : 8], 14 >_N$	{6,14,15,16}	data
$< [28], 13 >_Z < [12 : 28 : 8], 13 >_I < [4], 4 >_N < [12 : 28 : 8], 13 >_N$	{4,6,13,14,15,16}	data
$< [27], 12 >_Z < [11 : 27 : 8], 12 >_I < [11 : 27 : 8], 12 >_N$	{3,4,6,13,14,15,16}	data
$< [10 : 26 : 8], 11 >_Y < [10 : 26 : 8], 11 >_I < [10 : 26 : 8], 11 >_N$	{3,4,6,11,13,14,15,16}	data
$< [9, 25], 9 >_{P[0],P[1]} < [9, 25], 9 >_X < [17], 10 >_X < [9, 25], 9 >_I < [17], 10 >_I < [9, 25], 9 >_N < [17], 10 >_N$	{3,4,6,9,10,11,13,14,15,16}	data
$< [9, 25], 9 >_X < [17], 10 >_X < [9, 25], 9 >_I < [17], 10 >_I < [9, 25], 9 >_N < [17], 10 >_N$	{1,2,3,4,6,9,10,11,13,14,15,16}	memory
$< [8 : 24 : 8], 8 >_X < [8 : 24 : 8], 8 >_I < [8 : 24 : 8], 8 >_N$	{1,2,3,4,6,8,9,10,11,13,14,15,16}	data,control
$< [7 : 23 : 8], 7 >_I < [7 : 23 : 8], 7 >_N$	{1,2,3,4,6,7,8,9,8,10,13,14,15,16}	data
$< [6 : 30 : 8], 6 >_I < [6 : 30 : 8], 6 >_N$	{1,2,3,4,6,7,8,9,8,10,13,14,15,16}	solved queries

FIGURE 5.9. Implementing A&H's imprecise dynamic slicing algorithms.

5.3.2 Approximate dynamic slicing with TWPP+

In this subsection, we use TWPP+ representation to implement the approximate algorithms proposed in [1], identified as `approach` 1 and 2.

Imprecise algorithm 1: This method marks all executed nodes in the PDG during the execution. The backward traversal to identify the statements in the dynamic slice is allowed to visit only the marked nodes. These marked nodes are essentially the nodes with non-empty timestamp sets in our TWPP representation. Therefore in our implementation the backward traversal of a query through the timestamp annotated CFG is allowed to traverse only nodes that have a non-empty timestamp set. When a dependence is identified under such a traversal, the statement at which the dependence originates is added to the dynamic slice. In our example, all statements are executed. Therefore the dynamic slice is the same as a static slice, which contains all nodes except node 12.

Imprecise algorithm 2: This method marks all executed edges in the PDG during the execution. The backward traversal to identify the statements in the dynamic slice

is allowed to only traverse marked edges. The backward analysis uses timestamps to find dependences can carry out a similar traversal by ensuring that an edge from node n to node m is traversed only if the query at node m contains timestamp t and the timestamp $t-1$ is associated with node n. To find the memory dependence, all edges kept at a load instruction point are traversed back to include new nodes into the slice. Moreover since this algorithm does not distinguish between different timestamps corresponding to a node, when a dependence is found, and new queries are generated at a node, all timestamps of that node are included in the newly generated query for further propagation. In the example, we will be able to get the dynamic slice which includes all nodes except node 5 and 12.

5.4 Conclusion

As demonstrated by the three applications discussed in this chapter, the timestamped whole program path representation can be used in a wide range of areas. It is organized at different level such that different applications can find the required information more conveniently.

TWPP+ can be used to study the overall behavior of a program execution. By regrouping and sorting the memory dependence edges, redundant load and store instances are identified. A significant percentage of load instructions are highly redundant and could be further optimized to improve performance. With the timestamps, the exact execution order is maintained in the TWPP+ such that it is much faster to identify the frequency of some data flow facts at some program points with respect to the given whole program path. The TWPP+ representation can also be used as debug tool to create dynamic slices at any program execution point. Different slicing algorithms are simulated using this representation with different cost and slice accuracy tradeoff.

Chapter 6
Profiling dynamically allocated data objects

In the preceding chapters, a new representation was developed to compress both control flow and memory address profiles. It enables the application of whole program path in profile guided optimizations by speeding up the information retrieval at the analysis stage. On the other hand, with the rapid advances in both computer architecture and programming practice, *new* types of profiles are needed to explore *new* optimization opportunities and develop *new* types of optimization techniques. In the following three chapters, a new profiling framework is developed to discover runtime compression opportunities. Both software and hardware techniques are developed to exploit these opportunities.

Over the last decade, the memory and CPU performance gap has become a major performance bottleneck in modern computer architectures. Cache has been proposed as an effective component to bridge this gap. Since cache is usually much smaller than the main memory and the user space, it is very important to make good use of the cache memory in order to achieve good performance. Traditional approaches to improve cache performance such as doubling the size, increasing the associativity from hardware, or rearranging data objects or data fields within objects by compilers, do not change the data density in the cache. However, as we will see, a large percentage of the bits stored in both the cache and the main memory are redundant. By removing these redundant bits, more data items can be kept in a cache of given size and alleviate the memory bottleneck by reducing the number of cache misses.

The user space is divided into three areas: stack, globals and heap. The data structures allocated in different areas show different cache and memory accessing

behavior. Those allocated in stack usually have much better performance than the rest. Data layout optimizations can be used to optimize the behavior of global and heap data accesses. Those allocated in the global data space, even if they have bad performance, can be optimized well by existing compilers. However, those allocated in the heap have bad cache behavior and they are hard to optimize at compile time since they are allocated dynamically. The focus of this research is mainly on the dynamic data structures. In particular, new data compression techniques are designed to compress dynamically allocated data structures.

Before the design of a dynamic and effective data compression technique, we need to profile programs and collect information that would guide us in the development of new compression techniques. In this chapter, such a framework is presented for profiling dynamically allocated data objects. The framework allows us to analyze the value characteristics and lifetime of the dynamically allocated objects. More specifically, the framework allows us to carry out and answer the following questions.

- What data structures should be compressed?
- How should they be compressed?
- When should they be compressed?

The subsequent chapters present detailed software and hardware schemes respectively to remove redundancy in dynamically allocated heap data objects. While the software technique uses data compression transformations for redundancy removal, the hardware technique removes redundancy through a novel data cache design.

The rest of this chapter is organized as follows. The type based profiling technique is introduced in section 6.1. The experimental framework is presented in section 6.2. Results of studies aimed at answering the three questions listed above are presented in sections 6.3, 6.4 and 6.5 respectively. Section 6.6 concludes the chapter.

6.1 Type based profiling

Usually, a program contains a large number of objects of a given type and there are a number of fields within the given type. For example, all the nodes of a linked list are of the same structure with several fields: a pointer field to link the nodes together and some other data fields. Often there exists a significant degree of value similarity across the same fields from different nodes. Space requirements could be reduced by taking advantage of this similarity. However, a compression strategy that treats uniformly all fields in a type is too coarse and would miss many opportunities in practice. A new type-based profiling technique which collects the following information is proposed to solve this problem.

- The lifetime of each object and the total number of load and store accesses to this object are found through profiling. This information identifies the overall behavior of each dynamically allocated object.

- The value characteristics for each field of each type in the program are determined. This information is organized as the value range summary of all field instances. Value characteristic information can be collected at different granularity and it is possible to keep an additional list of most frequently use N values and their access frequencies.

To collect the above information, a straightforward approach is a complete instrumentation of all access points including the creation point at *malloc()*, the deletion point at *free()* and all load and store accesses. Although it is easy to trace at the high level the type information of objects through *malloc()* and *free()* function call points, a high level variable access can be translated to either a register access and a memory access. Since we are only interested in memory accesses, instrumentation at a high level is thus insufficient. One possible approach is to trace the load and store accesses by modifying the code generation part of the compiler but it is too expensive. To

minimize effort in modifying the compiler, a type-based profiling framework is proposed in this chapter with a combined approach of using high level instrumentation and lower level simulation.

6.2 Experimental framework

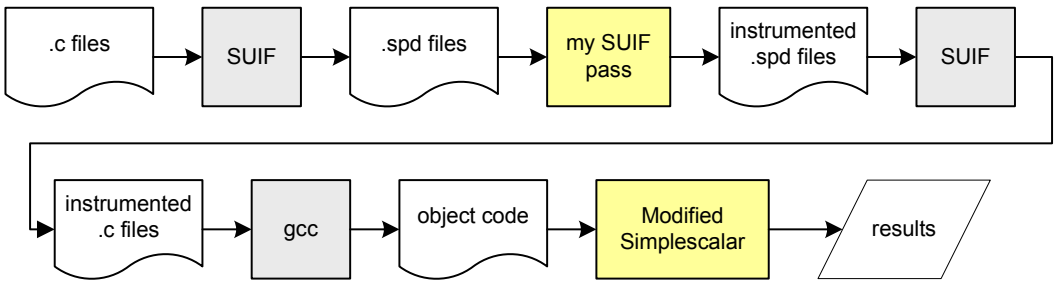


FIGURE 6.1. Type based profiling framework.

Figure 6.1 shows the framework which combines the use of SUIF 1.0 compiler [54] and Simplescalar simulator [10]. The original C programs are first converted to SUIF intermediate representations (IRs) by *scc*. A new pass is written to instrument these IRs and high level type of information is inserted into the instrumented code. The results of the new pass are still IRs and they are converted back to C programs by *s2c* (a conversion tool in SUIF). Then the instrumented version of C programs are compiled by *gcc* provided in Simplescalar and the MIPS-like executable code is generated. The Simplescalar simulator, which has been modified to process high level type information, is used to simulate the execution and collect profiles.

At the high level, the new SUIF pass instruments two kinds of program points and generates a type list for later reference.

- At each **memory allocation point**, a dummy instruction "asm("lw $0, T($26)")" is inserted before the function call *malloc*. Here the parameter "T" indicates the type index of the return memory address and two registers $0 and $26 are

explicitly used. $0 is a constant zero register and $26 is an operating system reserved register.

- At each **type casting point**, two dummy instructions "asm("lw $0, T1($26)")" and "asm("lw $0, T2($26)")" are inserted to indicate the types before and after casting.

- A **type list** records all types declared in the program. The mapping from each offset to its corresponding field is kept for each type. Given an offset for a type, it is possible to find its corresponding field and the type of the field.

The register "$26" is safe to use because as an operating system reserved register, the object code generated from a compiler such as *gcc* does not use this register. Since the object code is to be simulated rather than executed on a real machine, the simulator can catch these dummy instructions, nullify their effects except for extracting the type information they carry from the high level instrumentation. As a result, these dummy instructions have no effect on the program execution except the slots they take from the instruction cache. Because it is the lower level simulator that traces memory accesses, the high level code is inserted only at the malloc and type casting points. The number of these points is significantly smaller than the number of memory accesses. In this way the program's behavior is minimally affected by the instrumentation.

At the lower level, the simulator is modified to instrument one kind of program point and maintain two profiling data structures.

- **Memory access point.** Each memory access point, either a load or a store instruction, is traced but only the accesses to the interesting heap objects are further processed.

- **A B+ tree.** The simulator maintains a B+ tree to keep all nodes dynamically allocated from system malloc. When a new memory chunk is allocated, a new

record that contains the starting address, the size, and the result type of the memory chunk is inserted into this B+ tree. The starting address is gotten from the result of the call to *malloc*. The size and type information is obtained from the high level dummy instructions.

- **Field level compressibility list for each type.** The simulator maintains a list for each type. The field level compressibility information is maintained in this list to summarize the value properties for all instances.

Since nearly all data items are accessed at word level, an assumption is made to consider only the accesses at word size level from now on. Although there are accesses to fetch double precision floating point values, double word-sized values, or subword level values, the overall percentage of such accesses is usually very small. It is also possible to approximate each double-word access by two consecutive word accesses and each subword access by one word-sized access plus a subsequent bit extraction.

Once a memory access is traced by the simulator at runtime, a series of checks are performed as follows. First, the simulator searches the B+ tree and finds the record containing the address. Second, the offset is computed from the node's starting address. With simple calculation from the information in the type list, this memory address is mapped to a field and the corresponding field type information is determined. For example, suppose we are checking a memory address "0x10000108" and there is a node which indicates an array type with starting address "0x10000000". Furthermore, each item is 0x100 bytes long and is of the following type.

```
struct list_node {
  int value;
  struct list_node *prev,
  struct list_node *next;
  ...;
} *t;
```

The field being accessed is then mapped to the third field of the second item in the array. Finally, the value itself is checked to evaluate its compressibility and the field compressibility list is updated accordingly.

Next, let us discuss the experiments and their results in answering the *what*, *when* and *how* questions of data compression using this framework.

6.3 Selecting object types to compress

A program may contain multiple data types, each exhibiting different access patterns and different compressibility. The data types that an optimizing compiler should compress are those, by transforming which, positive impact on performance is expected. The fields in a data type should also be considered separately with the aim of maximizing benefit. Since different compression schemes might group fields differently, and thus affect the overall compressibility, this section will discuss how to separate and pick out different data structures for compression. The compression of different fields is left to subsequent sections.

As described, the collected profiles provide the information about the number of objects for each type and the information about the value ranges for all fields of a given type. Potential space savings can thus be calculated from this information. Experiments have been done to identify the appropriate data types to compress in SPEC95int benchmark suite. Most programs from this benchmark suite have at least one of the following properties.

- There are a set of similar types and a *generic type*. The *generic type* is used to build up the data structure but each node is of a specific type. Object instances are accessed by *type casting* to a specific type. Programs 130.li and 126.gcc, which themselves are compilers, exhibit this property. These programs build up a syntax tree for each function and each node in the tree could be of a specific type (e.g., for an expression, a FOR statement, or an IF statement, etc).

- The program first allocates a large chunk and starting address of the node is re-calculated (aligned) to a special address. Because of implicit address arithmetic, the compiler has difficulty in remapping the fields and their offsets. Program 124.m88ksim exhibits this property.

The above identified properties block further processing of SPEC95int benchmarks and we conclude that they are not good for automatic compression transformations.

The programs from Olden benchmark suites were further studied. Olden is a pointer intensive benchmark suite (Table 6.1). Although the types of a program are divided into several groups, each group has only one type. The clear type isolation provides good opportunities for compiler-based compression. The main data structures that were considered for compression are given in Table 6.1.

Program	Application	Main data structure
`bh`	Barnes & Hut N-body force computation algorithm	Heterogeneous tree
`bisort`	Bitonic Sorting	Binary Tree
`health`	Columbian health care simulation	Doubly- linked lists
`mst`	Minimum spanning tree of a graph	Heterogeneous tree
`perimeter`	Perimeter of regions in images	Quad-tree
`treeadd`	Recursive sum of values in a balanced B-tree	Binary tree
`tsp`	Traveling salesman problem	Balanced binary tree
`voronoi`	Computes the voronoi diagram of a set of points	Balanced binary tree

TABLE 6.1. Olden Benchmark Summary.

6.4 Choosing the compression scheme

A major challenge in the design of a compression scheme is to balance the dynamic compression costs and the benefits of compression. A dynamic compression scheme should be simple and fast for most if not all of the accesses. If applying traditional compression techniques (e.g., a dictionary based approach or Huffman coding), there

are at least two memory accesses: one to fetch the encoded data and the other to fetch the decoded data. As the cache and memory accesses are already the bottleneck and the major focus of applying compression dynamically is to reduce the total number of these accesses, these techniques are not appealing for dynamic compression. The new compression scheme that would be suitable to have in a dynamic environment should get all information about a value in one access for most if not all of the data accesses. Of course, subsequent computation to extract the decoded value might be inevitable. A logical comparison of the traditional and new compression schemes is shown in Figure 6.2.

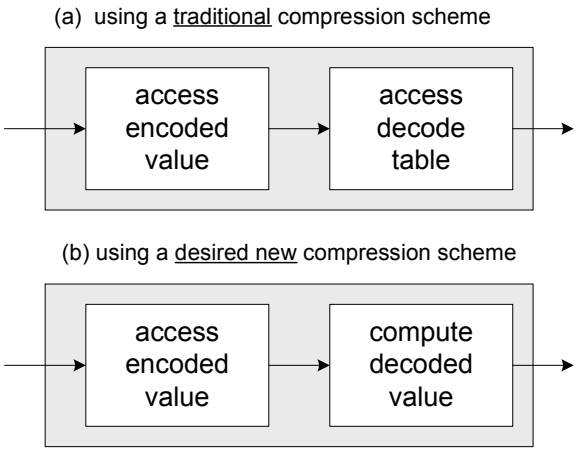
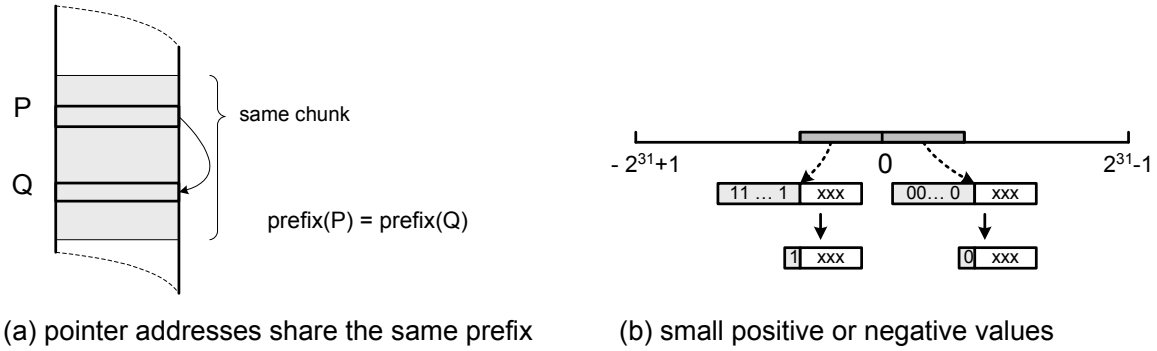


FIGURE 6.2. Access sequences with different compression schemes.

To design a compression scheme that can get all information from one memory access, no complicated encoding scheme should be used but rather we should discard directly the *redundant* bits from original word representation. The following two types of redundancy are identified (also see Figure 6.3).

- If a pointer is saved in a place that is close to the place it points to, the value of the pointer and the address of the pointer share the same prefix. Since the value is accessed always from its address, the prefix of the value can be considered as redundant as it can be constructed from its address easily. In this case, the



(a) pointer addresses share the same prefix (b) small positive or negative values

FIGURE 6.3. Representing a 32-bit value with fewer than 32 bits.

prefix bits of the pointer can be safely discarded.

- If a value is close to zero, the higher order bits are sign extensions and they are either all 0s or all 1s. In either case, there is no need to remember all these identical bits and thus the prefix bits are considered as redundant. Only the sign bit should be remembered and the rest can be safely discarded.

With the compression opportunities identified, one could dynamically discard all redundant bits and use the least possible bits to represent a value, or discard some redundant bits but use fixed number of bits to represent a value. Since data values change dynamically, the former strategy would bring too much complexity to dynamic memory management and thus is not used. For the latter, we need to choose the fixed number of bits based upon the cost-benefit analysis of using this fixed number of bits to carry out compression.

6.4.1 Potential savings in space due to redundancy removal

The benefits of a dynamic compression scheme come from the *space savings* and the corresponding *cache miss reduction* due to the space savings. As a result, the benefits estimation is based upon space savings and the experiment is designed as follows. The original scheme always represents a word-sized value with 32 bits. The new scheme

uses a fixed bit width L and if after removing the redundant prefix bits from a value's representation, the required number of bits is less that L, L bits are used to represent this value; otherwise, 32 bits are used to represent the value. The total number of bits required for representing values involved in all accesses for different values of L were collected.

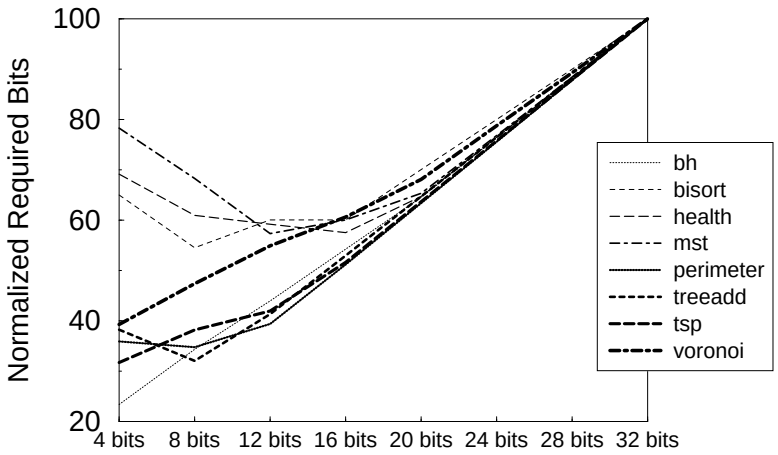


FIGURE 6.4. Required bits with fixed length.

The programs from the Olden suite with small inputs were executed to collect the profiling information. The results are shown in Figure 6.4 with the original required bits normalized as 100%. A smaller fixed length saves greater number of bits for each compressible value. However, if L is too small, the probability that a value will be represented using 32 bits is high. From the results in Figure 6.4, it can be seen that find some benchmarks achieve the best profiling results at the point with fixed 4 bits while some achieve the best results with 8 or 16 bits. However, if the fixed length is bigger than 16 bits, the required bits increase almost linearly for all benchmarks.

6.4.2 Potential costs of redundancy removal

Let us now study the cost of dynamic compression. The real cost is implementation dependent and a precise estimation can only be done when both the compression scheme and the lower level architecture are all well defined. A coarse estimation is

presented instead and it is sufficient to guide the design of the compression scheme. Since the benefits come from the compression of fields whose majority instances are compressible values, a successful compression scheme should speedup the accesses of compressible values; otherwise, the slowdown of majority accesses will downgrade significantly the overall performance. On the other hand, the accesses of incompressible values could be slower than those of compressible ones. Thus, the cost estimation is performed by answering the following question. If only accesses of incompressible values are slowed down, would the cost from accessing incompressible values be offset by the benefit obtained from accesses of compressible values? This in turn depends on the distribution of compressible and incompressible values. The results for Olden benchmark programs are shown in Figure 6.5. A memory access is considered to be a *fitting access* if its value can be represented by fixed 4, 8, 16 bits respectively. The results in Figure 6.5 show that more than half of the accesses could be expensive non-fitting accesses if using 4 bits. While with 16 bits, the percentages of fitting accesses are between 69% and 99%. As a result, *16-bit* is a cost-effective point and a good candidate to use.

Dynamic values change frequently and it is usually more expensive to convert a compressible value to an incompressible one, or *vice versa*. With dynamic expansion of values considered, a study of the benchmarks has been done from the storage point of view and the results are shown in Figure 6.6. All values are initially allocated with fixed lengths, 4 bits, 8 bits and 16 bits respectively. If a value changes from compressible to incompressible, it gets expanded and stays as an incompressible value from then onwards. Therefore later accesses will be fitting-accesses even if they are accessing the incompressible value. The results show that with a fixed 16-bit representation, the majority memory accesses fit this length and dynamic conversion is very infrequent.

From the above analysis and the results in estimated costs and benefits, we conclude that a well-balanced compression scheme should represent a 32 bits value with

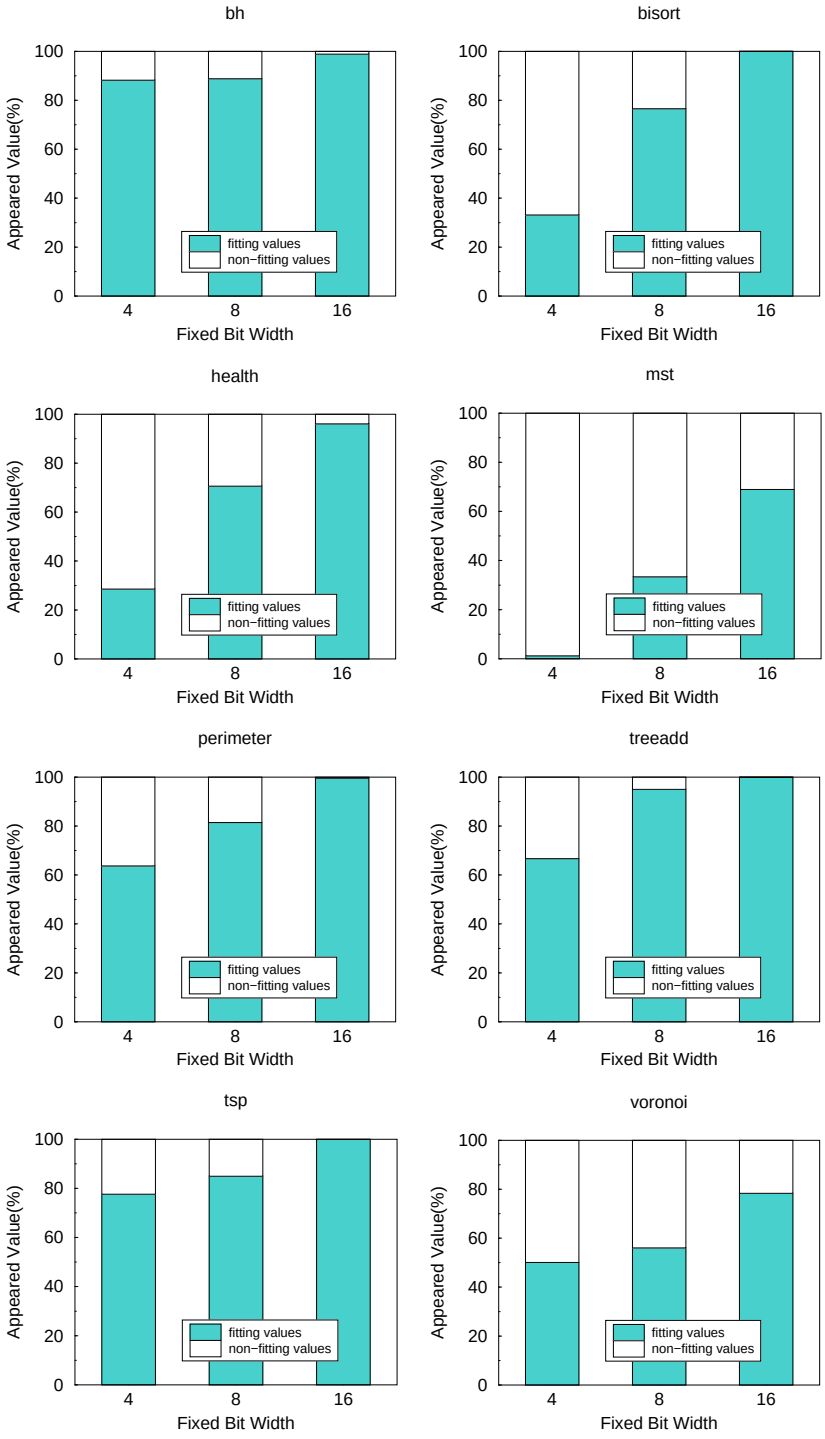


FIGURE 6.5. Distribution of values with fixed length.

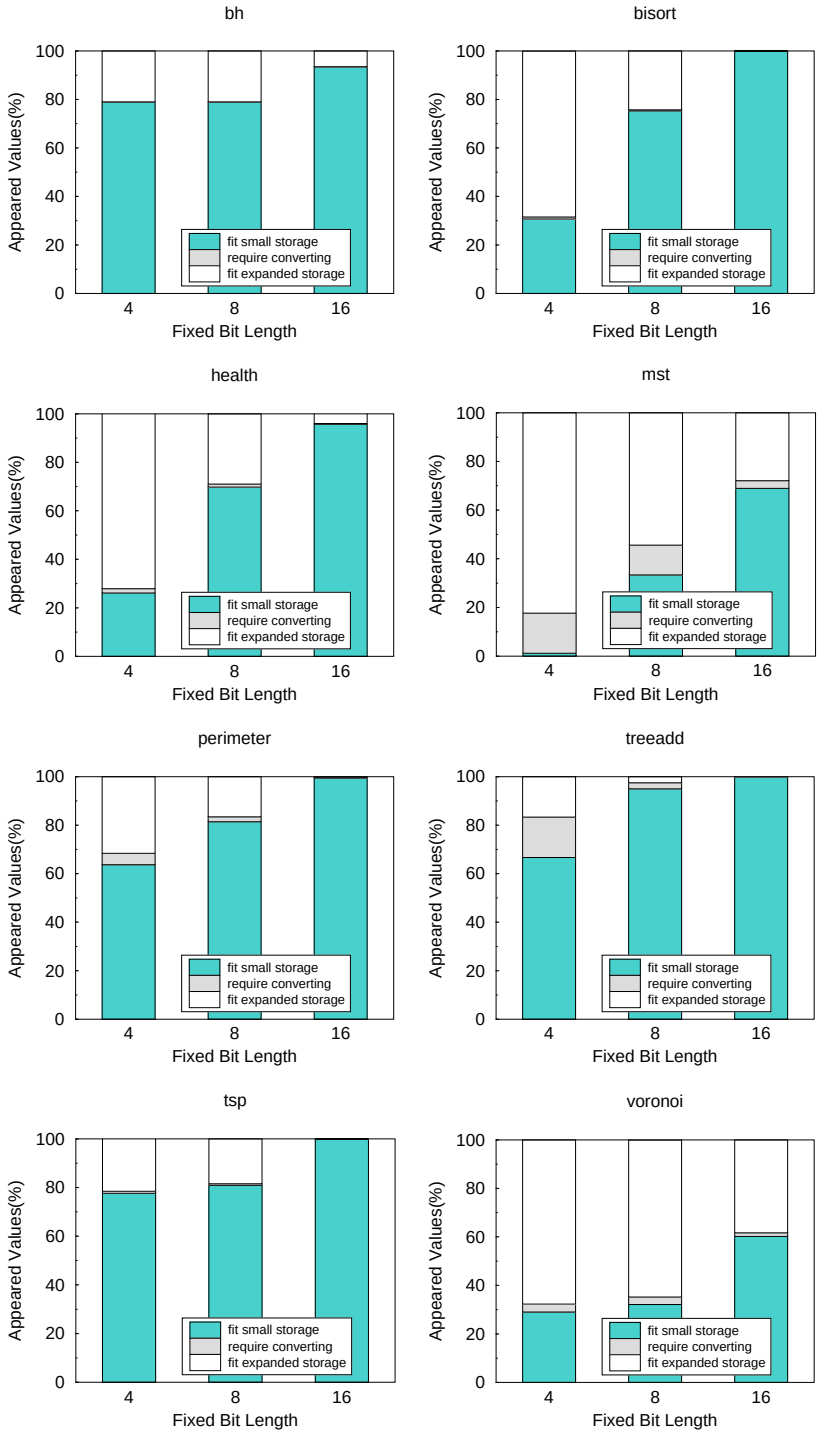


FIGURE 6.6. Distribution of values with fixed storage.

fixed 16 bits, allow dynamic expansion, but not allow dynamic shrinking.

6.5 Choosing the time for compression

The next problem to be considered is that of determining *when* these types should be compressed. The following three possible schemes are studied.

- Complete compression at the beginning. The simplest scheme is to compress all fields of a type at the beginning. If implementing in a compiler, it means that the memory layout of the type is reduced to half of its original size at compile time. It has the simplicity that the offset of each fields is known at compile time and code generation is therefore simplified.

- Selective compression at the beginning. Since different fields exhibit different compression opportunities and some fields such as floating point value fields are general hard to compress. Consider the cost the program has to pay at runtime to access incompressible fields, it is more preferable to compress only those highly compressible fields and leave the rest as they are. This scheme still has the property that the offsets of the fields are known at compile time.

- Compression after last write. Since the compressibility of a value can only be changed by a write operation, after the last write of a field, its representation is fixed and more aggressive compression scheme can be used and we do not have to worry that the values might change later. This completely eliminates the cost that a compression scheme has to pay to handle conversions from compressible values to incompressible ones.

Figure 6.7 shows the experimental results in studying the Olden benchmark programs under different compression time. For the selective compression scheme at the beginning, a field is chosen if 80% of its instances are compressible. The x axis of

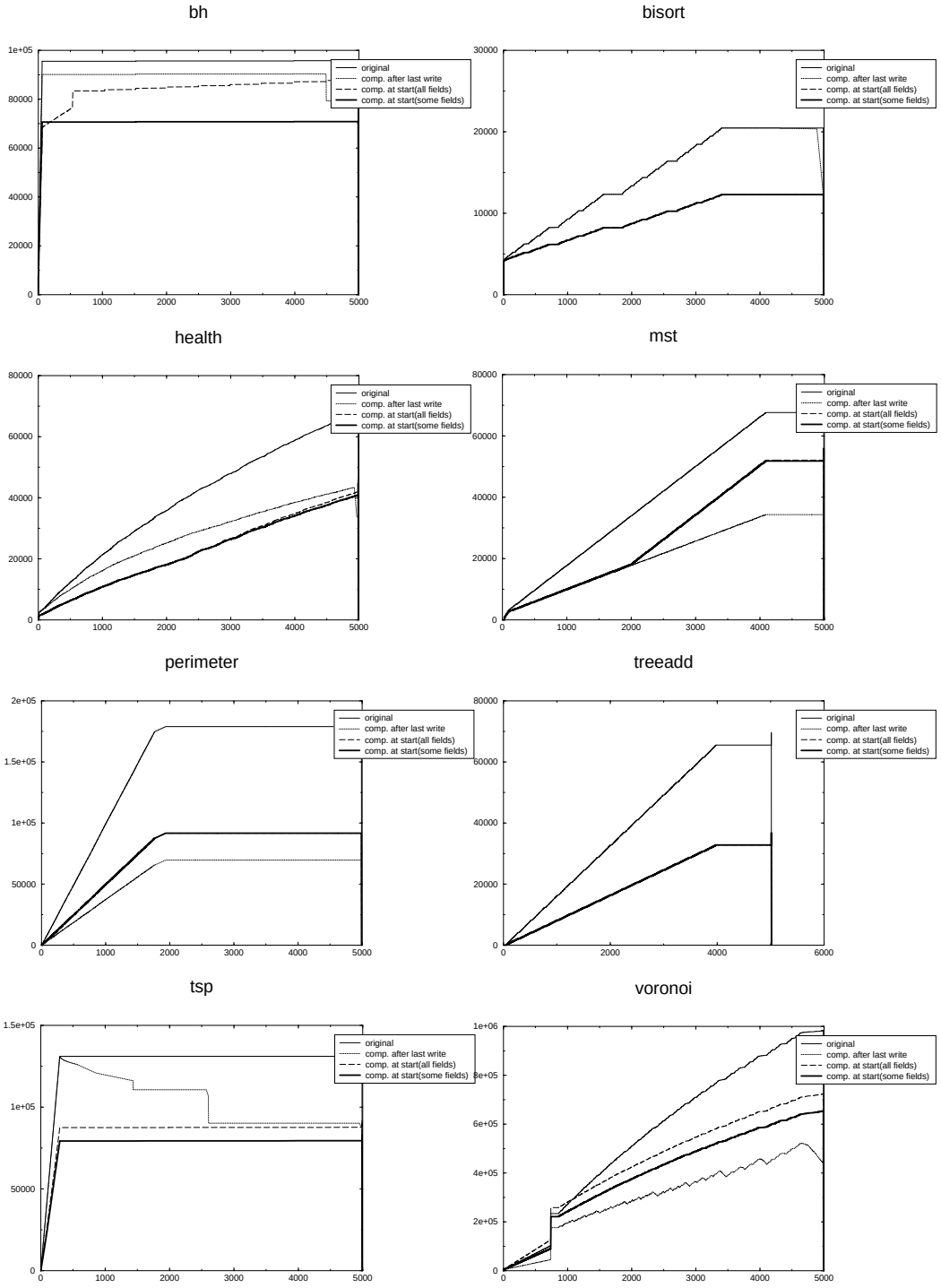


FIGURE 6.7. Deciding the time for compression.

the figure shows the execution time which has been normalized to 5000 units. The y axis shows the required heap space during the execution. "free()" is not considered during program execution and thus the memory requirements increase continuously and decrease to zero at the end of the execution.

The scheme that compresses an object after its last write can remove the dynamic conversion cost from compressible values to incompressible ones. It can also achieve best space savings for 4 out of 8 programs. Although it looks like an appealing approach, it is difficult to apply in practice. Usually at some program point, only a small number of nodes in a data structure are modified. However, the whole data structure might be traversed and nodes are dynamically selected for modification. It would be very expensive, and sometimes impossible, to predict the last write to a particular node.

Comparing the two schemes that compress data items at the beginning, the results show that although compressing all the fields achieves more space reduction at the beginning, it requires more data expansion during program execution. Thus eventually, it requires more space than the selective compression scheme. Moreover, the results show that selecing only highly compressible data fields for compression reduces both the space requirements and dynamic costs of accessing incompressible values. Overall, the scheme that compresses selected fields at the beginning achieves the best result for 5 out of 8 programs and almost the best for another 2 programs.

6.6 Conclusion

A type-based profiling framework is introduced in this chapter to explore the runtime value representation redundancy for Olden benchmark suites. It profiles each type in the program at field level and all object instances are checked to set up a cost-benefit model. The model is used in the design of new compression schemes. In particular, three important questions were answered. We decided to compress all objects of a

given type if there was no address arithmetic and no type casting. Promising fields were selected for compression at the beginning of the execution. A 32-bit value is represented using fixed 16 bits and while dynamic expansion is allowed, dynamic shrinking is not allowed. This chapter also identified two types of value representation redundancy from common prefix of pointer addresses and sign extensions of small values.

Chapter 7
Profile-guided data compression transformations

In the preceding chapter, a type based profiling framework was introduced to identify opportunities for reducing redundancy in the dynamic representation of values. Results of studies conducted using the profiling framework have identified the most important characteristics of a suitable compression scheme by answering the *what*, *how* and *when* questions. In this chapter, a concrete compiler based compression scheme will be introduced. The design details are consistent with the characteristics discussed in the preceding chapter.

A compiler based approach exploits the compression optimization opportunities through transformation. First, the data structures and types with compression opportunities are identified. Fields are packed in a way to achieve better cost-benefit ratio. Before code generation, the original memory layouts of these types are changed to compressed forms such that each node takes less space than before. Second, the original code sequence that accesses the compressed fields is converted to a new sequence to access the modified type with compression and decompression done dynamically. By reducing the value representation redundancy, dynamic resources such as cache and main memory space, memory bandwidth are utilized more effectively.

The rest of this chapter is organized as follows. The data compression transformations are introduced in section 7.1. The instruction and compiler support needed to perform these transformations are discussed in sections 7.2 and 7.3 respectively. Experimental results are shown in section 7.4. Finally, section 7.5 concludes the chapter.

7.1 Data compression transformations

From the discussion in the preceding chapter, it is known that many dynamic values exhibit value representation redundancy and the maximal cost-benefit ratio appears at about the point to represent values with fixed 16 bits. Therefore, employing a compiler transformation that replaces a 32-bit variable by a 16-bit variable and packs two variables into a single word is the logical choice. In the example below, a pointer field and a small value field are packed into a single 32-bit field *value_next*.

```
Original Structure:                 Transformed Structure:
  struct list_node {                    struct list_node {
    ...;                                        ...;
    int value;                                  int value_next;
    struct list_node *next;             } *t;
  } *t;
```

In this way, 4 bytes are saved from each node in the linked list. Although indicated by a type redeclaration, this transformation is not done at the source level and there is no need to generate the new declaration at source code level. Instead the optimizing compiler will change the memory layout before code generation and then generate new code sequences accordingly. As we see, there are two types of fields: pointer addresses and small value fields. They are handled differently through two types of data compression transformations.

Common-prefix transformation for pointer data. The pointer contained in the *next* field of the link list can be compressed under certain conditions. In particular, consider the addresses corresponding to an instance of *list_node* (`addr1`) and the *next* field in that node (`addr2`). If the two addresses share a common 17 bit prefix because they are located fairly close in memory, the *next* pointer is classified as compressible. In this case the common prefix from address `addr2` which is stored in the `next` pointer field is eliminated. The lower order 15 bits from `addr2` represent the representation of the pointer in compressed form. The 32 bit representation of a *next* field can be

reconstructed when required by obtaining the prefix from the pointer to the *list_node* instance to which the *next* field belongs.

Narrow data transformation for non-pointer data. Now let us consider the compression of the narrow width integer value in the *value* field. If the 18 higher order bits of this value are identical, that is, they are either all 0's or all 1's, it is classified as compressible. The 17 higher order bits are discarded and leaving a 15 bit entity. Since the 17 bits discarded are identical to the most significant order bit of the 15 bit entity, the 32 bit representation can be easily derived when needed by replicating the most significant bit.

Packing together compressed fields. The *value* and *next* fields of a node belonging to an instance of *list_node* can be packed together into a single 32 bit word as they are simply 15 bit entities in their compressed form. Together they are stored in *value_next* field of the transformed structure. The 32 bits of *value_next* are divided into two half words. Each compressed field is stored in the lower order 15 bits of the corresponding half word. According to the above strategy, bits 15 and 31 are not used by the compressed fields. Next the handling of incompressible data in partially compressible data structures is described. The implementation of partially compressible data structures requires an additional bit for encoding information. This is why fields are compressed down to 15 bit entities and not into 16 bit entities.

Partial compressibility. The basic approach is to allocate only enough storage to accommodate a compressed node when a new node in the data structure is created. Later, as the pointer fields are assigned values, it is checked to see if the fields are compressible. If they are, they can be accommodated in the allocated space; otherwise additional storage is allocated to hold the fields in uncompressed form. The previously allocated location is now used to hold a pointer to this additional storage. Therefore

for accessing incompressible fields the approach has to go through an extra step of indirection.

If the incompressible data stored in the fields is modified, it is possible that the fields may now become compressible. However, such checks are not carried out and instead the fields in such cases are left in uncompressed form. This is because exploitation of such compression opportunities can lead to repeated allocation and deallocation of extra locations if data values repeatedly keep oscillating between the compressible and incompressible kind. To avoid repeated allocation and deallocation of extra locations the approach is simplified so that once a field is assigned an incompressible value, from then on wards, the data in the field is always maintained in uncompressed form.

The most significant bit (bit 31) in the word is used to indicate whether or not the data stored in the word is compressed or not. It contains a 0 to indicate that the word contains compressed values. If it contains a 1, it means that one or both of values were not compressible and instead the word contains a pointer to an extra pair of dynamically allocated locations which contain the values of the two fields in uncompressed form. While bit 31 is used to encode extra information, bit 15 is never used for any purpose.

The example in Figure 7.1 illustrates the above method using an example in which an instance of *list_node* is allocated and then the *value* and *next* fields are set up one at a time. As we can see first storage is allocated to accommodate the two fields in compressed form. As soon as the first incompressible field is encountered additional storage is allocated to hold the two fields in uncompressed form. Under this scheme there are three possibilities which are illustrated in Figure 7.3. In the first case both fields are found to be compressible and therefore no extra locations are allocated. In the second case the *value* field, which is accessed first, is compressible but the *next* field is not. Thus, initially *value* field is stored in compressed form but later when the *next* field is found to be compressible, extra locations are allocated and both

fields are store in uncompressed form. Finally in the third case the *value* field is not compressible and therefore extra locations are allocated right away and none of the two fields are ever stored in compressed form.

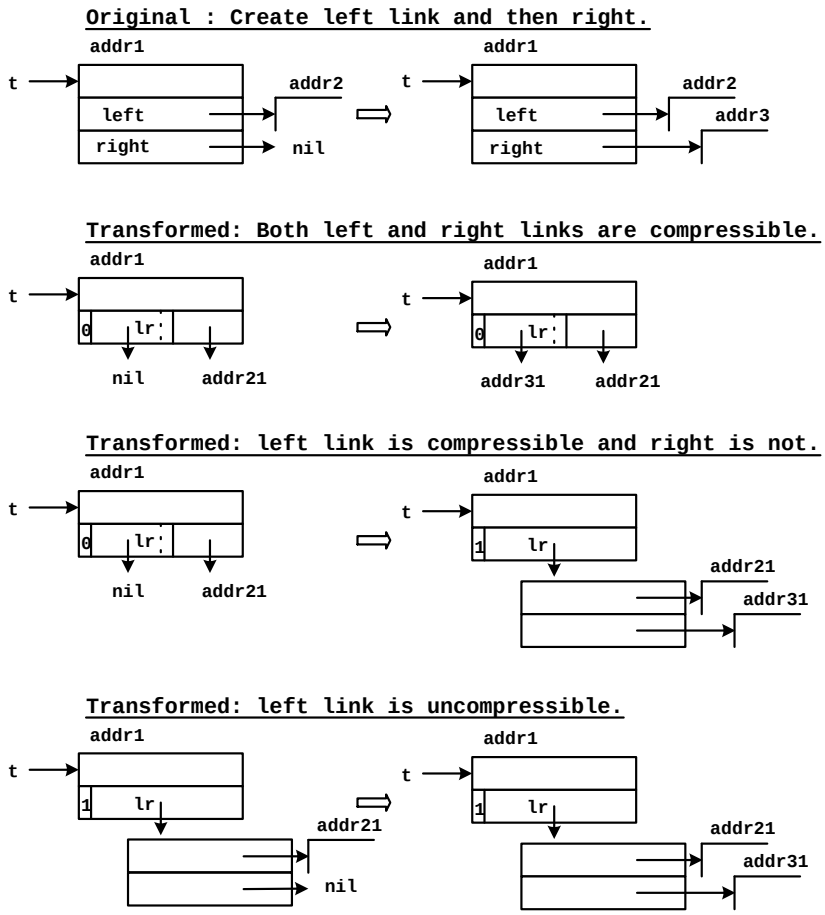


FIGURE 7.1. Dealing with incompressible data.

7.2 Instruction set support

Compression reduces the amount of heap allocated storage used by the program which typically improves the data cache behavior. Also if both the fields need to be read in tandem, a single load is enough to read both the fields. However, the manipulation of the fields also creates additional overhead. To minimize this overhead new RISC-

style instructions are designed. Six simple instructions have been designed of which three each are for pointer and non-pointer data respectively that efficiently implement common-prefix and narrow-data transformations. The semantics of the these instructions are summarized in Figure 7.2. These instructions are RISC-style instructions with complexity comparable to existing branch and integer ALU instructions. Let us discuss these instructions in greater detail.

Checking compressibility. Since we would like to handle partially compressible data, before actually compressing a data item at runtime, first a check is made to determine whether the data item is compressible. Therefore the first instruction type that is introduced allows efficient checking of data compressibility. Two new instructions have been designed and they are described below. The first checks the compressibility of pointer data and the second does the same for non-pointer data.

`bneh17 R1, R2, L1` – is used to check if the higher order 17 bits of `R1` and `R2` are the same. If they are the same, the execution continues and the field held in `R2` can be compressed; otherwise the branch is taken to a point where we handle the situation, by allocating additional storage, in which the address in `R2` is not compressible. The instruction also handles the case where `R2` contains a nil pointer which is represented by the value 0 both in compressed and uncompressed forms. Since 0 represents a nil pointer, the lower order 17 bits of an allocated address should never be all zeroes - to correctly handle this situation we have modified our `malloc` routine so that it never allocates storage locations with such addresses.

`bneh18 R1, L1` – is used to check if the higher order 18 bits of `R1` are identical (i.e., all 0's or all 1's). If they are the same, the execution continues and the value held in `R1` is compressed; otherwise the value in `R1` is not compressible and the branch is taken to a point where we place code to handle this situation by

allocating additional storage.

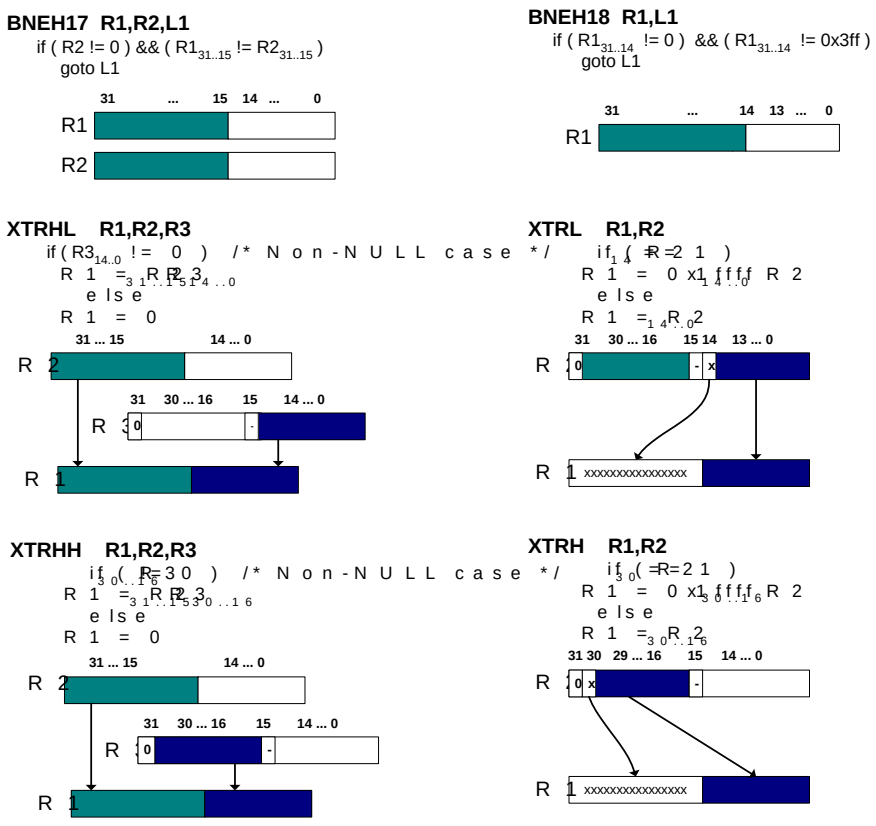


FIGURE 7.2. DCX instructions.

Extract-and-expand. If a pointer is stored in compressed form, before it can be dereferenced, its 32-bit representation must be reconstructed. Compressed non-pointer data should be handled similarly before its use. Therefore the second instruction type that is introduced carries out extract-and-expand operations. There are four new instructions that we describe below. The first two instructions are used to extract-and-expand compressed pointer fields from lower and upper halves of a 32-bit word respectively. The next two instructions do the same for non-pointer data.

`xtrhl R1, R2, R3` – extracts the compressed pointer field stored in lower order bits (0 through 14) of register `R3` and appends it to the common-prefix contained in

higher order bits (15 through 31) of `R2` to construct the uncompressed pointer which is then made available in `R1`. The case when `R3` contains a nil pointer is also handled. If the compressed field is a nil pointer, `R1` is set to nil.

`xtrhh R1, R2, R3` – extracts the compressed pointer field stored in the higher order bits (16 through 30) of register `R3` and appends it to the common-prefix contained in higher order bits (15 through 31) of `R2` to construct the uncompressed pointer which is then made available in `R1`. If the compressed field is a nil pointer, `R1` is set to nil.

The instructions `xtrhl` and `xtrhh` can also be used to compress two fields together. However, they are not essential for this purpose because typically there are existing instructions which can perform this operation. In the MIPS like instruction set that was used in this work this was indeed the case.

`xtrl R1, R2` – extracts the field stored in lower half of the `R2`, expands it, and then stores the resulting 32 bit value in `R1`.

`xtrh R1, R2` – extracts the field stored in the higher order bits of `R2`, expands it, and then stores the resulting 32 bit value in `R1`.

Next a simple example is given to illustrate the use of the above instructions. Let us assume that an integer field $t \rightarrow value$ and a pointer field $t \rightarrow next$ are compressed together into a single field $t \rightarrow value\_next$. In Figure 7.3(a) it is shown how compressibility checks are used prior to appropriately storing $newvalue$ and $newnext$ values in to the compressed fields. In Figure 7.3(b) we illustrate the extract and expand instructions by extracting the compressed values stored in $t \rightarrow value\_next$.

```
        ; $16 :  &t->value_next
        ; $18 :  newvalue
        ; $19 :  newnext

        ; branch if newvalue is not compressible
        bneh18  $18, $L1
        ; branch if newnext is not compressible
        bneh17  $16, $19, $L1
        ; store compressed data in t->value_next
        ori     $19, $19, 0x7fff
        swr     $18, 0($16)
        swr     $19, 2($16)
        j       $L2
$L1:
        ; allocate extra locations and store pointer
        ; to extra locations in t->value_next
        ; store uncompressed data in extra locations
        ...
$L2:  ...
```

(a) Illustration of compressibility checks.

```
    ; $16:  &(t->value_next)
    ; $17:  uncompressed integer t->value
    ; $18:  uncompressed pointer t->next

    ; load contents of t->value_next
    lw   $3,0($16)
    ; branch if $3 is a pointer to extra locations
    bltz $3,  $L1
    ; extract and expand t->value
    xtrl $17, $3
    ; extract and expand t->next
    xtrhh$18, $16, $3
    j    $L2
$L1:
    ; load values from extra locations
    ...
$L2:  ...
```

(b) Illustration of extract and expand instructions.

FIGURE 7.3. An example.

7.3 Compiler support

Similar to object layout optimization techniques, data compression transformations need to rearrange the fields in an object. A basic assumption of object layout transformations states that it is ensured by the programmer that application of layout transformations are safe (program correctness is ensured). Generally, if there is no address arithmetic and fields are accessed from their names, the assumption can be satisfied. Starting from this assumption, the optimizing compiler automatically transform the data types and generate corresponding code. The key aspects of the compiler transformation are discussed as follows.

Identifying fields for compression and packing. The candidate fields are classified from the type-based profiling described in the previous chapter. A field is identified to be *highly compressible* if 90% of the fields instances are compressible. A pointer value is considered as compressible if it shares the 17 bits prefix with its address and a small value is considered as compressible if the higher order 18 bits are the same.

The most critical issue is that of pairing compressed fields for packing into a single word. Based on the profiling information, fields are further categorized into *hot* fields and *cold* fields. With all categorized fields, there are two choices in packing. It is possible to pack *two hot fields* together if they are typically accessed in tandem. This is because in this situation a single load can be shared while reading the two values. It is also useful to compress any *two cold fields* even if they are not accessed in tandem. This is because even though they cannot share the same load, they are not accessed frequently. In all other situations it is not as useful to pack data together because even though space savings will be obtained, execution time will be adversely affected.

Ccmalloc vs malloc. `Ccmalloc` [17], a modified version of `malloc`, is used to carry out *storage allocation*. This form of storage allocation was developed by Chilimbi et

al. [17] and as described earlier it improves the locality of dynamic data structures by allocating the linked nodes of the data structure as close to each other as possible in the heap. Compared to system *malloc*, it has one more pointer parameter which indicates the parent node of the new node. The new node is allocated in the same cache line chunk as it parent node if there are still enough space to hold the new one. Otherwise, a new chunk is allocated. As a consequence, this technique increases the likelihood that the pointer fields in a given node will be compressible. Therefore it makes sense to use `ccmalloc` in order to exploit the synergy between `ccmalloc` and data compression.

Register pressure. Another issue that we consider in our implementation is that of potential increase in *register pressure*. The code executed when the pointer fields are found to be incompressible is substantial and therefore it can increase register pressure significantly causing a loss in performance. However, we know that this code is executed very infrequently since very few fields are incompressible. Therefore, in this piece of code we first free registers by saving values and then after executing the code the values are restored in registers. In other words, the increase in register pressure does not have an adverse effect on frequently executed code.

Instruction cache behavior and code size. The additional instructions generated for implementing compression can lead to an increase in *code size* which can further impact the *instruction cache* behavior. It is important to note however that a large part of the code size increase is due to the handling of the infrequent case in which the data is found not to be compressible. In order to minimize the impact on the *code size* we can share the code for handling the above infrequent case across all the updates corresponding to a given data field. To minimize the impact of the performance on the *instruction cache*, we can employ a code layout strategy which places the above infrequently executed code elsewhere and create branches to it and back so

that the instruction cache behavior for more frequently executed code is minimally affected. Our implementation currently does not support the above techniques and therefore we observed code size increase and degraded instruction cache behavior in our experiments.

Code generation. The remainder of the code generation details for implementing data compression are in most part quite straightforward. Once the fields have been selected for compression and packing together, whenever a use of a value of any of the fields is encountered, the load is followed by an extract-and expand instruction. If the value of any of compressed fields is to be updated, the compressibility check is performed before storing the value. When two hot fields that are packed together are to be read/updated, initially we generate separate loads/stores for them. Later in a separate pass, the later of the two loads/stores is eliminated whenever possible.

7.4 Implementation and experiments

7.4.1 Experimental setup

Techniques described have been implemented to evaluate their performance. The transformations have been implemented as part of the `gcc` compiler and the DCX instructions have been incorporated in the MIPS like instruction set of the superscalar processor simulated by `simplescalar` [10]. The evaluation is based upon six benchmarks taken from the *Olden* test suite which contains pointer intensive programs that make extensive use of dynamically allocated data structures.

In order to study the impact of memory performance we varied the input sizes of the programs and also varied the L2 cache latency. The programs were run for three input sizes – small (this is the standard input that is typically used to run the benchmark), medium and large (see Figure 8.10(a)). The cache organization of simplescalar is shown in Figure 8.10(b). There are first level separate instruction

and data caches (I-cache and D- cache). The lower level cache is a unified-cache for instructions and data. The L1 cache used was a 16K direct mapped cache with 9 cycle miss latency while the unified L2 cache is 256K with 100/200/400 cycle miss latencies. Our experiments are for an out-of-order issue superscalar with issue width of 4 instructions and the *Bimod* branch predictor.

Program	Application	small input	medium input	large input
`treeadd`	Recursive sum of values in a B-tree	20 1	21 1	22 1
`bisort`	Bitonic Sorting	32768 1	128000 1	312000 1
`tsp`	Traveling salesman problem	65536 1	131072 1	262144 1
`perimeter`	Perimeters of regions in images	12 1	13 1	14 1
`health`	Columbian health care simulation	3 2000 1	3 3000 1	3 4000 1
`mst`	Minimum Spanning tree of a graph	512 1	1024 1	2048 1

(a) Benchmarks and inputs used.

Parameter	Value
Issue Width	4 issue, out of order
Instruction cache	16K direct map
Icache miss latency	9 cycles
Level 1 data cache	16K direct map
Level 1 data cache miss latency	9 cycles
Level 2 unified cache	256K 2-way asso.
Memory latency (level 2 cache miss latency)	Configuration 1/2/3 = 100/200/400 cycles

(b) Cache configurations used.

FIGURE 7.4. Experimental setup.

7.4.2 Impact on storage needs

The transformations applied for each program and their impacts on node sizes are shown in Figure 7.5. In the first four benchmarks (`treeadd`, `bisort`, `tsp`, and `perimeter`), node sizes are reduced by storing pairs of compressed pointers in a single word. In the `health` benchmark a pair of small values are compressed together and stored in a single word. Finally, in the `mst` benchmark a compressed pointer and

Program	Transformation Applied	Node Size Change (bytes)
`treeadd`	CommonPrefix/CommonPrefix	from 28 to 20
`bisort`	CommonPrefix/CommonPrefix	from 12 to 8
`tsp`	CommonPrefix/CommonPrefix	from 36 to 32
`perimeter`	CommonPrefix/CommonPrefix	from 12 to 8
`health`	NarrowData/NarrowData	from 16 to 12
`mst`	CommonPrefix/NarrowData	from 16 to 12

FIGURE 7.5. Applied transformations.

a compressed small value are stored together in a single word. The changes in node sizes range from 25% to 33% for five of the benchmarks. Only in case of `tsp` is the reduction smaller – just over 10%.

The runtime savings in heap allocated storage are measured for each of the three program inputs. The results are given in Figures 7.6(a-c). The average savings are nearly 25% while they range from 10% to 33% across different benchmarks. Even more importantly these savings represent significant levels of heap storage – typically in megabytes. For example, the 33% storage savings for `treeadd` represents 4.2 Mbytes, 8.3 Mbytes, and 17 Mbytes of heap storage savings for small, medium and large program inputs respectively. It should also be noted that such savings cannot be obtained by other locality improving techniques described earlier [56, 35, 13, 17, 16].

From the results in Figure 7.6(a-c) another very important observation is made. The *extra locations* allocated when non-compressible data is encountered is non-zero for all of the benchmarks. In other words we observe that for none of the data structures to which our compression transformations were applied, were all of the instances of the data encountered at runtime actually compressible. A small amount of additional locations were allocated to hold a small number of incompressible pointers and small values in each case. Therefore the generality of our transformation which allows handling of *partially compressible* data structures is extremely important. If the application of compression was restricted to data fields that are always guaranteed

Program	Storage (bytes)		
	Original	Compressed nodes + Extra locations = Total	Space savings
`treeadd`	12582900	8388600 + 13440 = 8402040	33.2 %
`bisort`	786420	524280 + 25600 = 549880	30.1 %
`tsp`	5242840	4194272 + 6080 = 4200352	19.9 %
`perimeter`	4564364	3260260 + 5120 = 3265380	28.5 %
`health`	566872	509952 + 320 = 510272	10.0 %
`mst`	3414020	2367492 + 320 = 2367812	30.6 %
`average`			25.4 %

(a) Reduction in heap storage for small input.

Program	Storage (bytes)		
	Original	Compressed nodes + Extra locations = Total	Space savings
`treeadd`	25165812	16777208 + 26560 = 16803768	33.2 %
`bisort`	3145716	2097144 + 136320 = 2233464	29.0 %
`tsp`	10485720	8388576 + 12160 = 8400736	19.9 %
`perimeter`	9322572	6658980 + 10560 = 6669540	28.5 %
`health`	847584	762348 + 320 = 762668	10.0 %
`mst`	13643780	9453572 + 320 = 9453892	30.7 %
`average`			25.2 %

(b) Reduction in heap storage for medium input.

Program	Storage (bytes)		
	Original	Compressed nodes + Extra locations = Total	Space savings
`treeadd`	50331636	33554424 + 51260 = 33605684	33.2 %
`bisort`	3145716	2097144 + 204160 = 2301304	26.8 %
`tsp`	20971480	16777184 + 23040 = 16800224	19.9 %
`perimeter`	20332620	14523300 + 23680 = 14546980	28.5 %
`health`	1128240	1014804 + 320 = 1015124	10.0 %
`mst`	54550532	37781508 + 320 = 37781828	30.7 %
`average`			24.9 %

(c) Reduction in heap storage for large input.

FIGURE 7.6. Impact on storage.

to be compressible, no compression would have been achieved and therefore no space savings would have resulted.

Program	Code Size (bytes)		
	Original	Transformed	Increase
`treeadd`	5480	6376	16.4%
`bisort`	11944	16720	40.0%
`tsp`	18280	19172	4.9%
`perimeter`	14976	18160	21.3%
`health`	15952	21324	33.7%
`mst`	12768	14136	10.7%
`average`			21.1%

(a) Code size before linking.

Program	Code Size (bytes)		
	Original	Transformed	Increase
`treeadd`	228360	228444	0.04%
`bisort`	257552	257572	0.01%
`tsp`	238004	238448	0.18%
`perimeter`	233736	238340	1.97%
`health`	256608	257200	0.23%
`mst`	232296	232440	0.06%
`average`			0.41%

(b) Code size after linking.

FIGURE 7.7. Impact on object code size.

The increase in code size caused by compression transformations was also measured (see Figures 7.7). The increase in code size prior to linking is significant while after linking the increase is very small since the user code is small part of the binaries. However, the reason for significant increase in user code is because each time a compressed field is updated, our current implementation generates a new copy of the additional code for handling the case where the data being stored may not be compressible. In practice it is possible to share this code across multiple updates.

Once such sharing has been implemented, the increase in the size of user code will also be quite small.

7.4.3 Impact on execution time

Based upon the cycle counts provided by the simplescalar simulator we studied the changes in execution times resulting from compression transformations. The impact of input size and L2 latency on execution times was also studied. Let us examine the results in Figure 7.8(a) – these results are for L2 cache latency of 100 cycles. The reduction in execution times in comparison to the original programs which use `malloc` range from 3% to 64% while on an average the reduction in execution time is around 30%. The reductions in execution times increase gradually with the input size.

The execution times are compared with versions of the programs that use `ccmalloc`. The new approach outperforms `ccmalloc` in five out of the six benchmarks (our version of `mst` runs slightly slower than the `ccmalloc` version). On an average it outperforms `ccmalloc` by nearly 10%. Our approach outperforms `ccmalloc` because once the node sizes are reduced, typically greater number of nodes fit into a single cache line leading to a low number of cache misses. Additional runtime overhead is incurred in form of extra instructions needed to carry out compression and extraction of compressed values. However, this additional execution time is more than offset by the time savings resulting from reduced cache misses; thus leading to overall reduction in execution time.

It should be pointed out that the use of special DCX instructions was critical in reducing the overhead of compression and extraction. Without DCX instructions the programs would have ran significantly slower. The average reduction in execution times, in comparison to original programs, dropped from 30% to 12.5%. Instead of an average reduction in execution times of 10% in comparison to `ccmalloc` versions

of the program we observed an average increase of 9% in execution times.

The experiments of Figure 7.8(a) were also repeated for higher L2 cache latencies. The results are presented in Figures 7.8(b-c). As the latency of L2 cache is increased, compression outperforms `ccmalloc` by a greater extent. The graph in Figure 7.8(d) plots the average reduction in execution time that compression provides over `ccmalloc` for the different cache latencies. As it can be seen, on an average, compression reduces the execution times by 10%, 15%, and 20% over `ccmalloc` for L2 cache latencies of 100, 200, and 400 cycles respectively. These numbers also improve gradually with input size. In summary our approach provides large storage savings and significant execution time reductions over `ccmalloc`.

7.4.4 Impact on power consumption

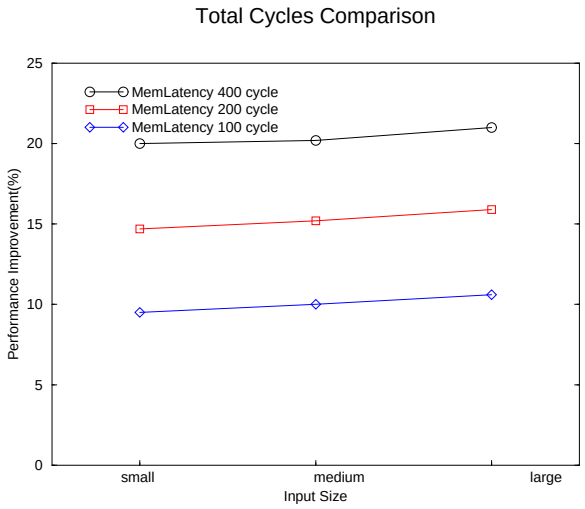
Experiments have also been done to compared the power consumption for the compression based programs with that of the original programs and `ccmalloc` based programs (see Figures 7.9(a-d)). These measurements are based upon the `Wattch` [9] system which is built on top of the `simplescalar` simulator. These results track the execution time results quite closely. The average reduction in power consumption over the original programs is around 30% which increases gradually with the size of the input. The graph in Figure 7.8(d) plots the average reduction in power dissipation that compression provides over `ccmalloc` for the different cache latencies. As we can see, on an average, compression reduces the power dissipation by 5%, 10%, and 15% over `ccmalloc` for L2 cache latencies of 100, 200, and 400 cycles respectively. These numbers further improve gradually as the input size is increased.

7.4.5 Impact on cache performance

Finally, Figure 7.10 presents the impact of compression on cache behavior, including I-cache, D-cache and unified L2 cache behaviors. As expected, the I-cache performance

Program	Input Size	Configuration 1		Configuration 2		Configuration 3	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	58.8 %	81.0 %	62.1 %	73.2 %	66.6 %	65.4 %
	medium	58.8 %	81.0 %	62.0 %	73.2 %	66.6 %	65.4 %
	large	58.7 %	81.0 %	62.0 %	73.2 %	66.5 %	65.4 %
`bisort`	small	75.3 %	73.9 %	62.8 %	58.6 %	48.6 %	42.9 %
	medium	69.3 %	69.5 %	54.8 %	53.4 %	40.7 %	38.7 %
	large	67.7 %	64.9 %	51.9 %	48.1 %	36.5 %	32.8 %
`tsp`	small	97.3 %	99.7 %	96.5 %	99.7 %	95.3 %	99.8 %
	medium	97.0 %	99.7 %	96.2 %	99.7 %	94.9 %	99.8 %
	large	96.8 %	99.5 %	95.9 %	99.6 %	94.5 %	99.6 %
`perimeter`	small	75.1 %	91.8 %	73.8 %	87.1 %	72.2 %	81.6 %
	medium	76.3 %	92.3 %	74.8 %	87.7 %	72.9 %	82.1 %
	large	77.6 %	93.1 %	76.0 %	88.5 %	73.9 %	82.9 %
`health`	small	83.4 %	94.6 %	83.6 %	91.3 %	83.7 %	89.3 %
	medium	68.1 %	95.5 %	66.3 %	93.1 %	65.3 %	91.8 %
	large	62.1 %	95.8 %	60.0 %	93.8 %	58.8 %	92.6 %
`mst`	small	35.7 %	102.2 %	28.3 %	101.7 %	23.2 %	101.2 %
	medium	37.8 %	102.1 %	31.2 %	101.6 %	26.8 %	101.0 %
	large	36.6 %	102.2 %	30.7 %	101.6 %	26.6 %	101.0 %
`average`	small	70.9 %	90.5 %	67.9 %	85.3 %	64.9 %	80.0 %
	medium	67.9 %	90.0 %	64.2 %	84.8 %	61.2 %	79.8 %
	large	66.6 %	89.4 %	62.7 %	84.1 %	59.5 %	79.0 %

(a) Change in cycle counts

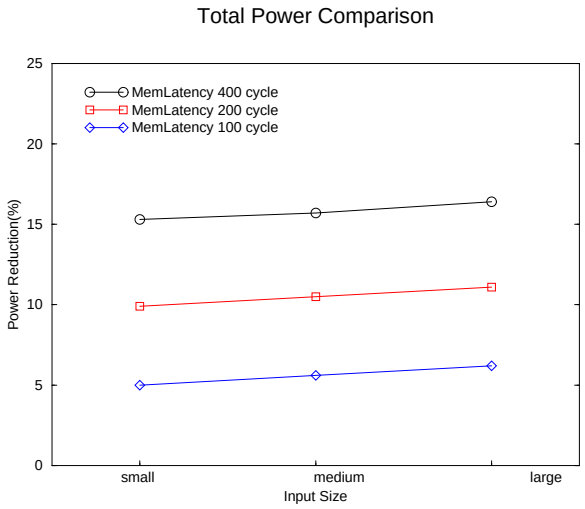


(b) Compression vs `ccmalloc`.

FIGURE 7.8. Change in execution time due to data compression.

Program	Input Size	Configuration 1		Configuration 2		Configuration 3	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	60.9 %	89.7 %	62.5 %	83.4 %	65.0 %	75.5 %
	medium	60.9 %	89.7 %	62.5 %	83.4 %	65.0 %	75.5 %
	large	60.9 %	89.7 %	62.5 %	83.4 %	65.0 %	75.5 %
`bisort`	small	77.9 %	79.2 %	67.6 %	66.0 %	54.9 %	51.1 %
	medium	73.7 %	75.9 %	61.3 %	61.6 %	47.9 %	46.8 %
	large	73.3 %	72.1 %	59.5 %	56.9 %	44.5 %	41.3 %
`tsp`	small	97.1 %	99.8 %	96.5 %	99.9 %	95.6 %	99.9 %
	medium	96.7 %	99.8 %	96.1 %	99.8 %	95.1 %	99.9 %
	large	96.7 %	99.7 %	96.0 %	99.7 %	94.9 %	99.7 %
`perimeter`	small	75.9 %	95.1 %	74.7 %	91.6 %	73.1 %	86.8 %
	medium	77.1 %	95.6 %	75.8 %	92.2 %	74.1 %	87.4 %
	large	78.6 %	96.4 %	77.2 %	93.1 %	75.2 %	88.2 %
`health`	small	90.0 %	101.4 %	87.8 %	95.9 %	86.2 %	92.1 %
	medium	73.5 %	101.1 %	69.5 %	96.6 %	67.1 %	93.8 %
	large	66.8 %	100.9 %	62.7 %	96.8 %	60.2 %	94.3 %
`mst`	small	38.9 %	104.5 %	31.5 %	103.8 %	25.7 %	102.9 %
	medium	40.5 %	104.2 %	33.9 %	103.4 %	28.9 %	102.5 %
	large	39.7 %	104.2 %	33.5 %	103.4 %	28.7 %	102.5 %
`average`	small	73.4 %	95.0 %	70.1 %	90.1 %	66.7 %	84.7 %
	medium	70.4 %	94.4 %	66.5 %	89.5 %	63.0 %	84.3 %
	large	69.3 %	93.8 %	65.2 %	88.9 %	61.4 %	83.6 %

(a) Change in power consumption.



(b) Compression vs `ccmalloc`.

FIGURE 7.9. Impact on power consumption.

is degraded due to increase in code size caused by our current implementation of compression. However, the performances of D-cache and unified cache are significantly improved. This improvement in data cache performance is a direct consequence of compression.

Program	Input Size	I-cache		D-cache		Unified-cache	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	105.2 %	104.8 %	62.2 %	60.4 %	85.1 %	49.7 %
	medium	106.4 %	105.5 %	61.5 %	59.7 %	85.0 %	49.7 %
	large	107.3 %	104.7 %	60.0 %	59.8 %	84.9 %	49.7 %
`bisort`	small	153.3 %	155.9 %	65.0 %	58.7 %	16.2 %	16.8 %
	medium	228.2 %	234.1 %	68.7 %	63.1 %	15.5 %	16.6 %
	large	228.2 %	234.1 %	47.3 %	38.3 %	7.4 %	7.0 %
`tsp`	small	5.0 %	120.5 %	70.1 %	90.3 %	84.1 %	100.1 %
	medium	4.0 %	122.1 %	66.0 %	94.0 %	84.4 %	100.1 %
	large	3.6 %	124.9 %	62.4 %	84.3 %	84.4 %	100.1 %
`perimeter`	small	145.1 %	86.0 %	69.1 %	71.3 %	67.1 %	67.0 %
	medium	205.1 %	83.5 %	68.9 %	70.9 %	67.0 %	66.8 %
	large	321.8 %	78.0 %	69.1 %	70.3 %	67.0 %	66.8 %
`health`	small	122.2 %	112.1 %	82.2 %	96.2 %	41.6 %	62.3 %
	medium	133.8 %	116.6 %	82.2 %	97.8 %	46.4 %	67.3 %
	large	144.8 %	120.7 %	82.1 %	98.6 %	51.9 %	71.1 %
`mst`	small	26.6 %	61.6 %	41.0 %	100.9 %	16.2 %	100.0 %
	medium	16.8 %	48.2 %	49.0 %	96.3 %	21.3 %	100.0 %
	large	13.8 %	42.7 %	33.2 %	94.8 %	21.5 %	100.0 %
`average`	small	92.9 %	106.8 %	64.9 %	79.6 %	51.7 %	66.0 %
	medium	115.7 %	118.3 %	66.0 %	80.3 %	53.3 %	66.8 %
	large	136.6 %	117.5 %	59.0 %	74.3 %	52.8 %	65.8 %

FIGURE 7.10. Change in cache misses - configuration 1.

7.5 Related work

Recently there has been a lot of interests in exploiting narrow width values to improve program performance [9, 64, 61]. However, our work focuses on pointer intensive applications for which it is important to also handle *pointer data*. A lot of research has been conducted on development of locality improving transformations for dynamically allocated data structures. These transformations alter object layout and placement

to improve cache performance [56, 17, 13]. However, none of these transformations result in space savings.

Existing compression transformations [53, 18] rely upon compile time analysis to prove that certain data items do not require a complete word of memory. They are applicable only when the compiler can determine that the data being compressed is *fully compressible* and they only apply to *narrow width non-pointer* data. In contrast, our compression transformations apply to *partially compressible* data and, in addition to handling narrow width non-pointer data, they also apply to *pointer data*. The approach introduced in this chapter is not only more general but also simpler in one respect. It does not require compile-time analysis to prove that the data is always compressible. Instead simple compile-time heuristics are sufficient to determine that the data is likely to be compressible.

ISA extensions have been developed to efficiently process narrow width data including Intel's MMX [44] and Motorola's AltiVec [57]. Compiler techniques are also being developed to exploit such instruction sets [31]. However, the instructions introduced in this chapter are quite different from MMX instructions because both partially compressible data structures and pointer data must be handled.

7.6 Conclusion

In this chapter, two types of data compression transformations are introduced to apply data compression techniques to compact the sizes of dynamically allocated data structures. These transformations result in large space savings and also result in significant reductions in program execution times and power dissipation due to improved memory performance.

An attractive property of these transformations is that they are applicable to partially compressible data structures. This is extremely important because according to our experiments, while the data structures in all of the benchmarks studied

in this chapter are very highly compressible, they always contain small amounts of incompressible data.

This approach is applicable to a more general class of programs than existing compression techniques: it can compress pointers as well as non-pointer data; and it can compress partially compressible data structures. Finally the DCX ISA extensions have been designed to enable efficient manipulation of compressed data. The same task cannot be carried using MMX type instructions. The main contribution of this work is that data compression techniques can now be used to improve performance of general purpose programs and therefore it takes the utility of compression beyond the realm of multimedia applications.

Chapter 8
Exploiting value representation redundancy in hardware

Data compression transformations were introduced in the preceding chapter to exploit the value representation redundancy which was discovered from the type based profiling framework introduced in chapter 6. Both profiles and semantic information are used to select the fields for compression and pack them together. Cache performance is improved due to improved data locality. However, the approach also has some limitations. Source code has to be available in order to perform profiling, analyses and transformations. Thus, it is not applicable if only binary code is available. Moreover, restrictions such as address arithmetic and type casting may prohibit the application of these transformations (e.g., for SPEC benchmarks). In order to overcome the above drawbacks, a hardware-based approach for exploiting compression is considered in this chapter. This hardware approach does not analyze or transform the program and thus is applicable to all programs, including SPEC benchmarks which could not be handled by the data compression transformations.

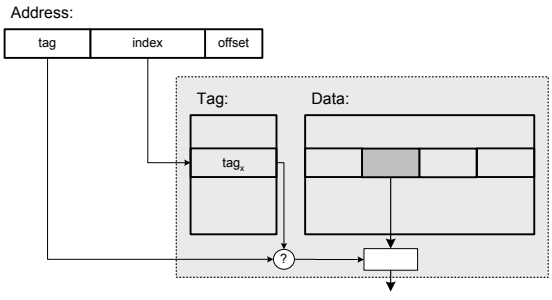


Figure 8.1. Memory address and cache access.

To study the potential of this approach, the characteristics of values involved

in word-sized accesses from the cache (Figure 8.1) were studied for programs from Olden, SPEC 95int, and SPEC 2000int benchmark suites. These values are divided into three categories: *compressible small values* – these are values whose higher order 18 bits are all zeros or all ones; *compressible address values* – these are values that share the same 17-bit prefix with their addresses; and *incompressible values* – these are all remaining values that are accessed. The results are summarized in Figure 8.2. On an average, 59% of dynamic appeared values are compressible and can be represented by less than or equal to 16 bits. Note that even though compiler based approach could not handle SPEC benchmarks, the data belonging to these programs is still highly compressible.

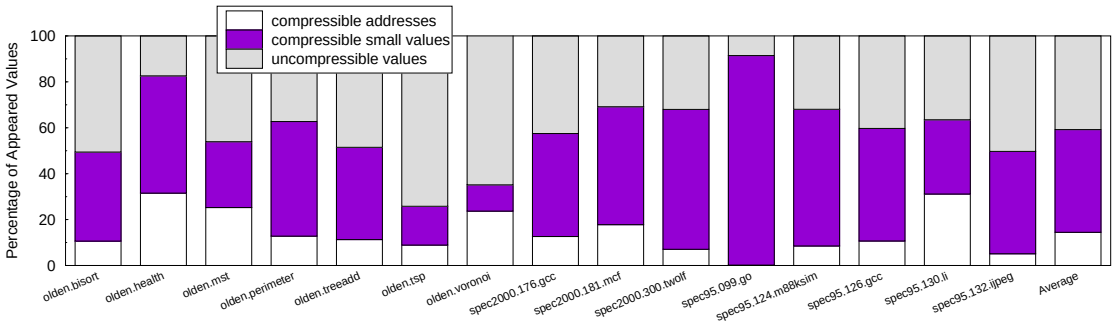


FIGURE 8.2. Values encountered during program execution.

Given the fact that the value representation redundancy exists uniformly across a spectrum of benchmark programs, it is useful to design a hardware approach which skips the complexity of compile-time analysis and transformation, and takes advantage of the value representation redundancy observed through profiling directly. In this chapter, a new cache design is proposed to exploit value representation redundancy. Values are stored in compressed form and the storage that is freed by this process is used to enable a novel style of cache line prefetching.

The rest of the chapter is organized as follows. The compression cache design is discussed in section 8.1. The implementation and experimental results are given in section 8.2. Related work will be discussed in section 8.3. Section 8.4 summarizes the

chapter.

8.1 Compression enabled partial cache line prefetching

First the representation of compressible values in hardware is given and then it is shown how the cache performance can be improved by enabling prefetching of partial caches lines. The description of how the cache is accessed and maintained dynamically is also given.

8.1.1 Value representation in hardware

As already discussed, in many cases, values can be represented by their lower order 16 bits as shown in Figure 8.3(a)(b). Figure 8.3(a) shows that the prefix of a pointer value could be discarded if it shares the same prefix with the memory address where the value is stored. Figure 8.3(b) shows that the prefix of a small value could be discarded if these bits are sign extensions.

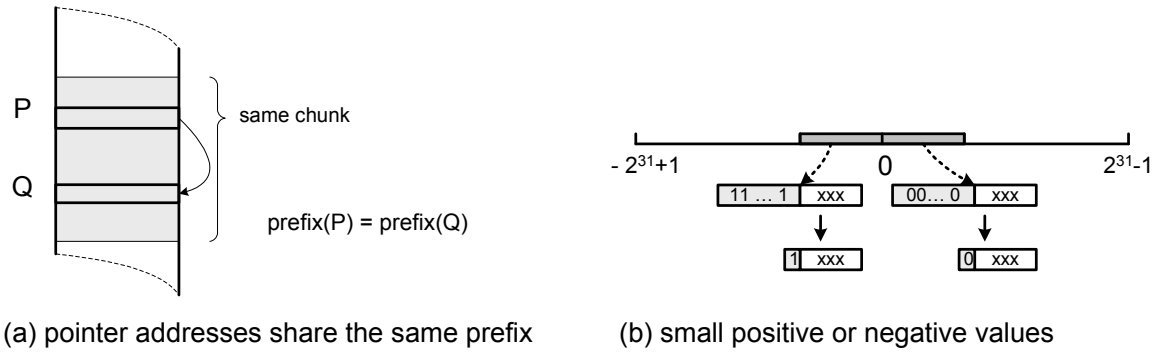


FIGURE 8.3. Representing a 32-bit value with fewer than 32 bits.

Since dynamically, both compressible and incompressible values will be encountered. When compressible values are represented in their compressed formats, a method is required to distinguish compressible values from incompressible ones. In addition, to reconstruct the original values at runtime, we must know whether they represent compressed addresses or compressed small values.

To help distinguish these cases, we have a 32-bit value compressed to 15 bits instead and use the 16th bit to tells its type (shown by "VT" in Figure 8.4). Similarly, we need one more bit to tell whether the word contains compressed or uncompressed values (shown by "VC" in Figure 8.4). However this bit is not stored as part of the value representation but stored in the cache as flags and will be discussed in more detail in cache design section.

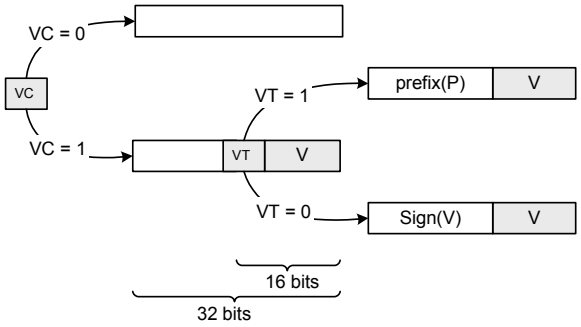


FIGURE 8.4. Representing compressed values in hardware.

8.1.2 Partial cache line prefetching

Hardware techniques for prefetching cache lines [49, 47] have been proposed to improve cache performance in high performance systems. If a cache line l is not in the cache, a memory access m for a word in l results in a cache miss. Prefetching loads the line l into the cache before m is encountered. By the time m is encountered later, l is already in the cache and there is no cache miss. In this way, prefetching hides the long cache miss latency. The problem with prefetching is that it greatly increases the memory traffic. Although it is a very effective technique for high performance systems with big memory bandwidth, the significant increase in memory traffic restricts its application to other systems such as embedded systems.

By exploiting the dynamic value representation redundancy, we can perform hardware prefetching with no increase in memory traffic. Our method fetches compressible values into the cache and stores the values in the cache in compressed formats. By

having a compressible word represented by 16 bits, a significant part of the cache space is spared. Let us consider the example shown in Figure 8.5 where it is assumed three out of four words are compressible in each cache line. The saved space in each cache line ($0.5 \times 4\ bytes/word \times 3\ words = 6\ bytes$) is not enough to hold another cache line. Therefore, we choose to prefetch only part of another line.

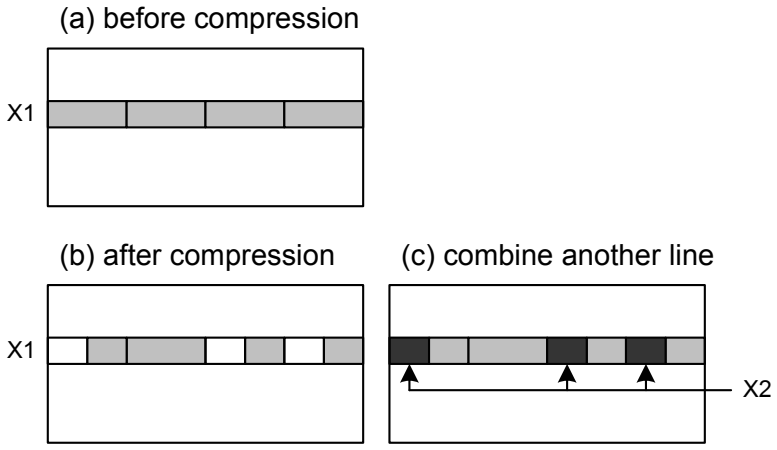


FIGURE 8.5. Compressing data in the cache to hold more words.

Let us consider the situation shown in Figure 8.5. If the compressible words from another cache line with corresponding offsets are prefetched, then three additional compressible words can be stored which covers 7 out of 8 words from two cache lines. On the other hand, if the incompressible words are fetched, we need two unused half-word-sized spots to store all bits of a prefetched word and some indexing space to indicate its order. The 6 bytes of available space can only store one more word from the prefetched line. Therefore a design is developed to only prefetch compressible values from another line.

The example in Figure 8.6 illustrates how compression enabled prefetching can enhance performance. Figure 8.6(b) shows a code fragment that traverses a link list whose node structure is shown in Figure 8.6(a). The memory allocator would align the address allocation and each node takes one cache line (we assume 16 bytes per line cache). There are 4 fields among which two are pointer addresses, one is a type

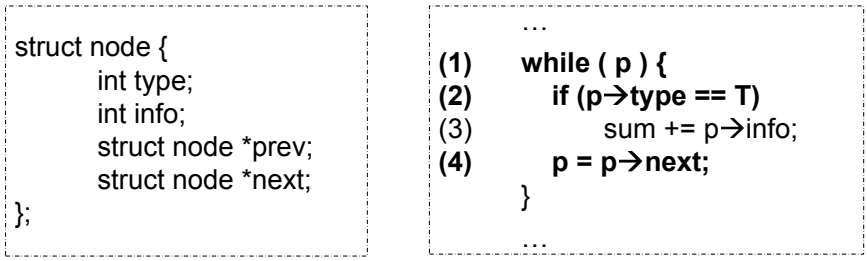


FIGURE 8.6. Dynamic data structure declaration.

field and the other one contains a large value. Except this large information value field, the other three fields are identified as highly compressible fields. The sample code shown in Figure 8.6(b) calculates the sum of the information field for all nodes of type T. Without cache line compression, each node takes one cache line. To traverse the list, the next field is followed and a new node is accessed.

A typical access sequence for this piece of code would generate a new cache miss at statement (2) for every iteration of the loop (see Figure 8.7(a)). All compression accesses to other fields in the same node fall into the same cache line and thus are all cache hits as shown in Figure 8.7(b). However, if all compressible fields are compressed, a cache line would be able to hold one complete node and three fields from another node. Now an access sequence will have cache hits at statements (2) and (4) plus a possible cache miss at statement (3). The partial cache line prefetching can improve performance in two folds. First, if the node is not of the type T, we do not need to access the large information field. This saves a cache miss. Second, even in the case we do need to access it, the cache miss happens at statement (3). Although the new and old scheme generate the same number of cache misses, the miss at statement (3) is not on the critical program execution path which is "(1)(2)(4)" and it has less impact on the performance.

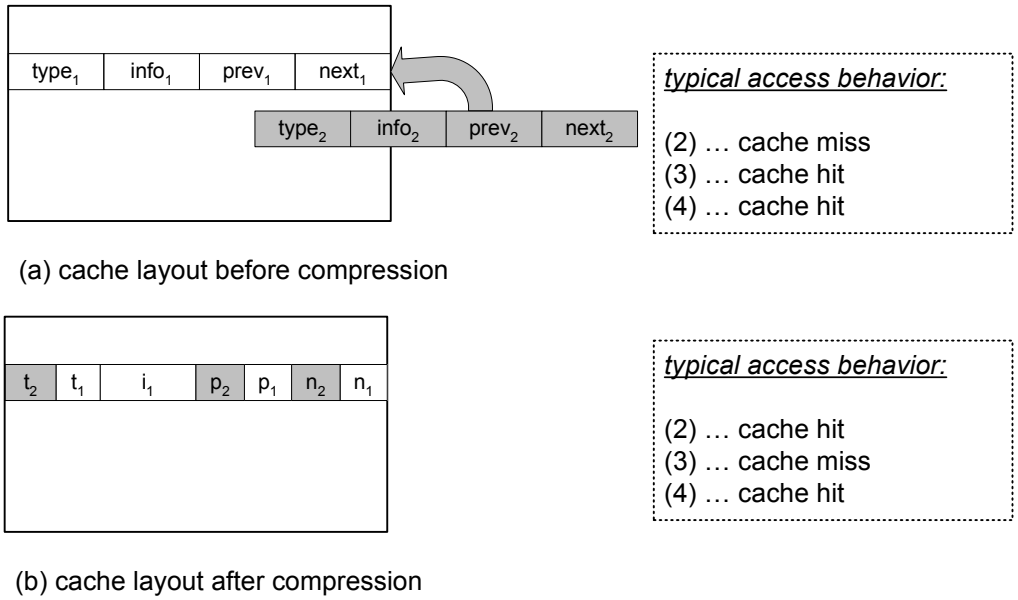


FIGURE 8.7. Cache layout before and after compression.

8.1.3 Cache design details

The new cache design can be implemented in either a single or a multiple level cache hierarchy. A two level cache hierarchy shown in Figure 8.8 is used and the compression enabled partial cache line prefetching is employed in both caches.

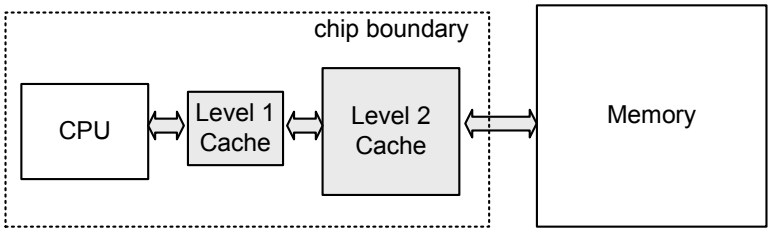


FIGURE 8.8. Two level compression cache design.

The compression scheme used is described as before. A value is compressible if it satisfies either of the following two conditions.

- If the 18 higher order bits are all ones or all zeros, the 17 higher order bits are discarded.

- If the 17 higher order bits are the same as those of the value's address, the 17 higher order bits are discarded.

The physical cache line at each level can potentially hold the contents from two lines, identified as the *primary* cache line and the *affiliated* line. The primary cache line is defined as the line mapped to this set in a normal cache of the same size and associativity. Its affiliated cache line is the unique line that is calculated through a single operation as shown below.

$$< Tag_{affiliated}, Set_{affiliated} >=< Tag_{primary}, Set_{primary} > \oplus mask$$

where the mask is a predefined value. The mask is chosen to be 0x1 which means the primary and affiliated cache lines are consecutive lines of data. Thus, given a cache line, it has two possible places to stay in the cache, its primary location and an affiliated location. Our cache access and replacement policy described later ensure that at most one copy of a cache line is kept in the cache at any time.

In a standard two level cache hierarchy, the requests from the upper level are cache line based. For example, if there is a miss at the first level cache, a request for the whole line is issued to the second level cache. In the compression cache design, the requested line might stay as an affiliated one in the second level cache and thus contains only partial data. To maximize the benefits from partially prefetched cache line, there is no need to get a complete line as long as the requested data item can be found. So the compression cache design still keeps the requests to the second level cache as word based and a cache hit at the second level cache only returns a partial cache line. The returned line might be placed as a primary line or an affiliated line. In either case, flags are needed to indicate whether a word is available in the cache line or not. A flag PA (Primary Availability) for the primary cache line is associated with one bit for each word and another flag AA (Affiliated Availability) for the affiliated cache line is provided. As discussed, a value compressibility flag (VC) is used to identify if a value is compressible or not. For the values stored in the primary line,

a one-bit VCP flag is associated for each word. On the other hand, if a value can appear in the affiliated line, it must be compressible and thus no extra flag is needed for these values. The design details of the first level compression cache are shown in Figure 8.9.

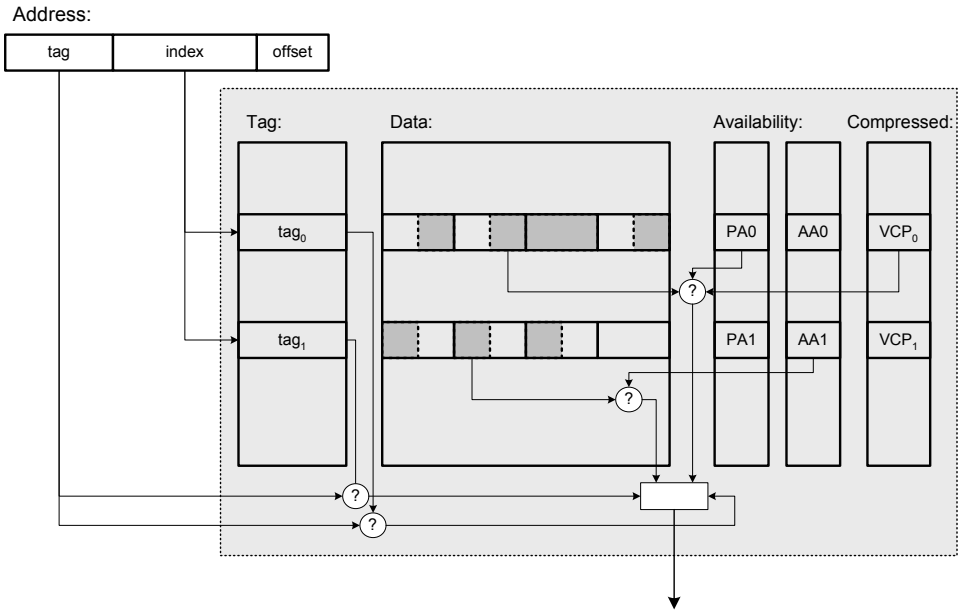


FIGURE 8.9. Compression cache.

8.1.4 Dynamic value representation

It happens only in the best case that both the primary and the affiliated lines are fully compressible. Normally, some words from one or both lines are not compressible. In those cases, priority is given to hold the words from the primary line. Thus, the primary line can always find the place to save the value while the affiliated line only keeps a word, if this word is compressible, and the word at the same offset from the primary line is also compressible.

At runtime, if a value changes from compressible to incompressible, a place to store the value must be found. There are two possibilities. If the value is to be stored in a primary cache line, the corresponding word from the affiliated cache line is kicked

out. The affiliated line is written back to lower level memory hierarchy if it is dirty. If the value is to be stored in an affiliated cache line, the affiliated line is moved to its primary place. The other line which stays in its primary place is kicked out and written back if that line is dirty.

8.1.5 Cache access policy

There are three cache interfaces to consider: CPU/L1 cache, L1/L2 cache and L2 cache/Memory. For a cache access from CPU to L1 cache, the set index of its primary cache line is extracted, the least significant bit is flipped to access its affiliated line. Both lines are accessed simultaneously. If the tag matches either of them, and its corresponding availability bit is set, the word is extracted, extended and returned to the CPU. For a cache access from L1 cache to L2 cache, if the accessed word is available in L2, it is a cache hit and only the available words are returned. For an access from L2 cache to memory, both the primary and the affiliated lines are fetched. However, before returning the data, the cache lines are compressed and only available places from the primary line are used to store the compressible items from the affiliated line. The memory bandwidth is still the same as before.

For both L1 and L2 cache, when a new cache line arrives, the prefetched affiliated line is discarded if it is already in the cache (it must be in its primary place in this situation). When a new cache line replaces an existing cache line, the affiliated place of the existing cache line is checked to see if the tag matches. If yes, the compressible words are filled into the available spots of its affiliated place. However, if the line is dirty, we still write back the content and only keep a clean partial copy in its affiliated place.

8.2 Implementation and experiments

8.2.1 Experimental setup

The compression enabled partial cache line prefetching has been implemented and evaluated using Simplescalar 3.0 [10]. We use a two level cache hierarchy: separate 8K first level data and instruction caches and a unified 64K level two cache. For the baseline configuration, L1 data cache is direct mapped and unified L2 cache is two-way set associative. Our compression cache is designed on top of the baseline configuration, with the ability to match its affiliated cache line. Since the proposed cache design doubles the number of lines searched in comparison to the baseline configuration, comparison is also made to a cache of higher associativity: *a 2-way set associative L1 cache plus a unified 4-way set associative L2 cache*. They are of the same size as the baseline configuration. Other parameters are all the same and summarized in Figure 8.10. A spectrum of programs from Olden [14], SPEC95, and SPEC2000 [50] benchmark suites are used.

Parameter	Value
Issue Width	4 issue, out of order
I cache	8K direct mapped (64 bytes/line)
I cache miss latency	10 cycles
L1 data cache	8K direct mapped
L1 data cache miss latency	10 cycles
L2 unified cache	64K 2-way (128 bytes/line)
Memory latency	100 cycles (L2 cache miss latency)

FIGURE 8.10. Baseline experimental setup.

8.2.2 Overall performance

Figure 8.11 shows the overall performance comparison with the baseline and the higher associativity cache configurations. The results are normalized with respect to

the baseline cache performance. Smaller numbers mean better results.

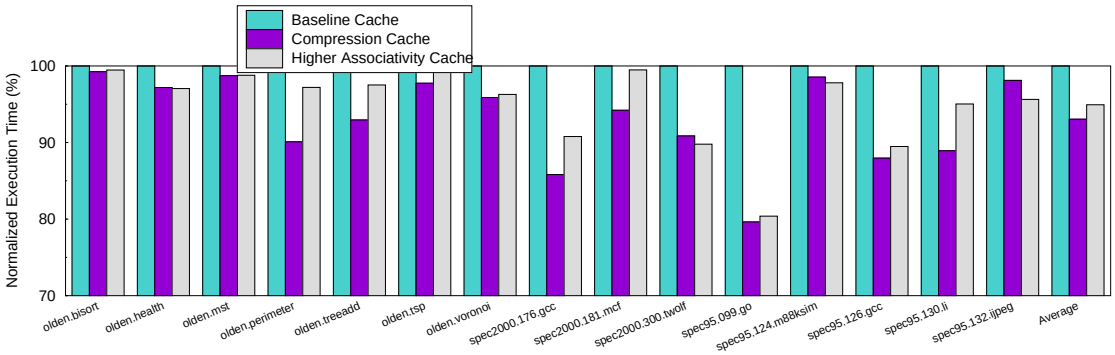


FIGURE 8.11. Performance comparison.

The newly designed cache gives consistently better results than the baseline cache. On an average, programs run about 7% faster. Speedup comes from the fact that unlike many other prefetching schemes which save the prefetched data into the cache and have the possibility of polluting the cache line, the compression cache never kicks out a cache line if the baseline cache does not have to replace it with the same access sequence. As a result, the cache miss rate can reduce but never increase. In many cases, the new design even outperforms the higher associativity cache configuration. The reason is that although higher associativity cache has a better replacement policy, the proposed cache can keep more data. For example, in a two-way set associative cache, 2 cache lines form one set can hold the contents from two lines at most while in the compressed directed mapped cache, two cache lines can potentially hold the contents of 4 lines. While the proposed design may have higher number of *conflict misses*, than the higher associativity cache, it may have fewer *capacity misses* if the data items are highly compressible. On an average, execution time is 2% faster than that of a higher associativity cache.

8.2.3 Cache miss comparison

The comparison results of L1 and L2 cache misses for different configurations are shown in Figure 8.12 and Figure 8.13 respectively. As we can see, through prefetching,

the compression cache greatly reduces the cache misses compared to the baseline configuration.

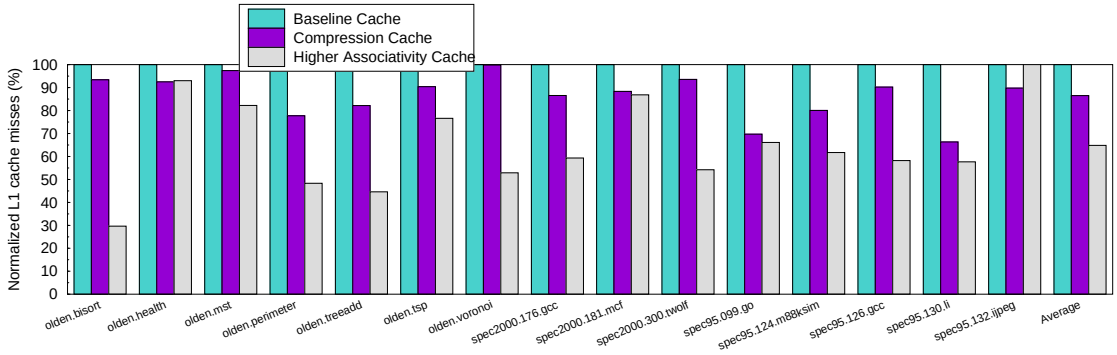


FIGURE 8.12. Comparison of L1 cache misses.

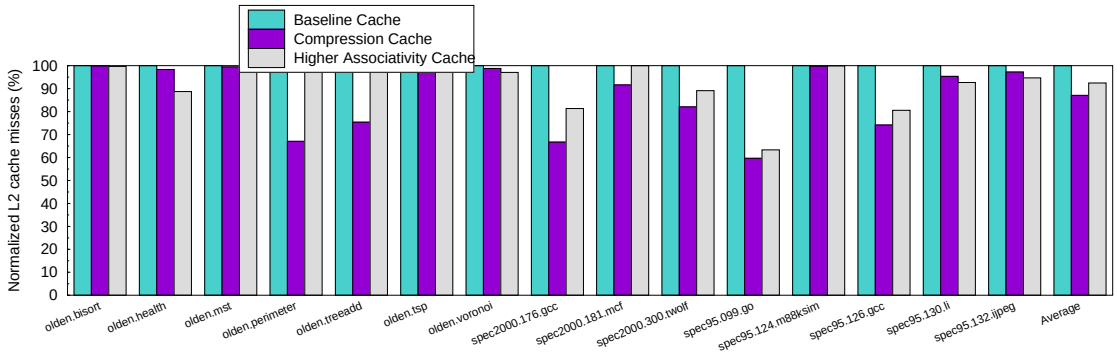


FIGURE 8.13. Comparison of L2 cache misses.

An interesting phenomenon observed is that although in many cases the compression cache has more L1 cache misses than the higher associativity cache configuration, it still achieves better overall performance. For example, for 130.li from SPECint95, although the new cache design has more L1 and L2 cache misses than the higher associativity cache, 6% improvement in performance over the higher associativity cache is observed. As was mentioned in the previous sections, this suggests that different cache misses have different performance impacts, i.e. some cache misses hurt the performance more than other cache misses.

To further analyze this phenomenon, we carried out additional experiments. Given a set of memory access instructions m, the importance of this set is defined as the

percentage of total executed instructions that directly depend on m. In case that m is the set of all cache miss instructions from a program execution, its importance parameter indicates how many dependent instructions are blocked by the cache misses. A higher number means that the cache misses block more instructions and thus hurt the performance more. The method to approximately compute this percentage is shown as follows. According to Amdahl's law, we have

$$\begin{aligned} Speedup_{overall} &= \frac{Execution_{old}}{Execution_{new}} \\ &= \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \end{aligned}$$

$$\therefore Fraction_{enhanced} = \frac{Speedup_{enhanced}(1 - \frac{1}{Speedup_{overall}})}{Speedup_{enhanced} - 1}.$$

In the Simplescalar simulator, without speculative execution, the memory address generated and their accesses are affected by the following factors: the executable program, the input, the seed for the random generator. If all these factors are fixed, two runs with different cache configurations will generate exactly the same instruction execution sequence as well as the memory address access sequence. Thus, by varying only the cache miss penalty and running the program twice, we would observe the same number of cache misses happen at the same instructions. Moreover, given this fixed set of instructions that have cache misses, their directly dependent instructions are also fixed. As we know, by shortening the miss penalty, the main change to the execution is the reduced dependence length from a cache miss instruction to its directly dependent instructions, the enhanced fraction could thus be considered as the percentage of the instruction that are directly depending on these cache misses.

Now, for different cache configurations, this fraction is computed as follows. First, the cache miss latency is reduced in half, which means $Speedup_{enhanced}$ =2. Second, the overall performance speedup is measured, which is $Speedup_{overall}$. It is computed

from the total number of cycles before and after changing the miss penalty. Now, the value of $Fraction_{enhanced}$ can be obtained. The results are plotted in Figure 8.14.

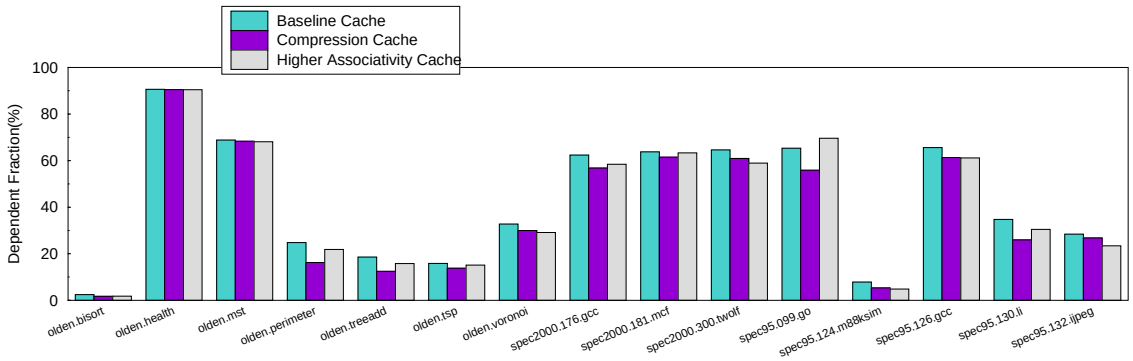


FIGURE 8.14. The estimation of cache miss importance.

From the comparison results for different benchmark programs shown in Figure 8.14, it can be seen that the compression cache reduces the importance of the cache misses for most benchmarks. For the benchmarks that are slower than the higher associative cache, it is seen that they have larger importance parameters. For the benchmarks with significant importance reduction, further study of the average ready queue length, when there is at least one outstanding cache miss, was carried out. The queue length increase of our compression cache over the higher associativity cache was studied. The results are shown in Figure 8.15. The results indicate that the average queue length is improved by up to 78% for these benchmarks. This parameter tells us when there is a cache miss in the new cache design, the pipeline still has a lot of work to do.

In summary, the cache misses that are encountered in the proposed compression cache design are less important in comparison to both the baseline and the higher associativity cache configurations.

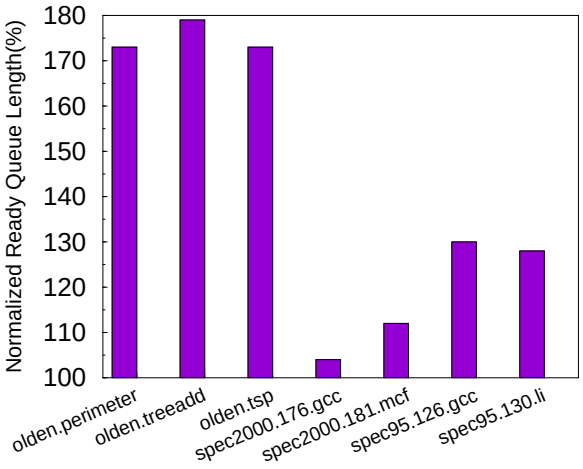


FIGURE 8.15. Average miss cycle ready queue length.

8.2.4 Memory traffic

The partial prefetching of the next cache line is enabled only in the case that there are available spots in the primary cache line and the corresponding affiliated words are also compressible. By combining the lower order bits from two words, and using the same memory bandwidth, more data items are effectively transmitted. So the memory traffic is not increased. Actually, the overall memory traffic is reduced because of the reduction in the second level cache misses. The only situation that may cause increased memory traffic happens if a store instruction writes to the primary place or the affiliated place changes a compressible value to an incompressible one. Either it will generate a cache miss if writing to the affiliated place, or kick out a (dirty) affiliated line. In either cases, the memory traffic would increase. However, since this happens infrequently, a net reduction in memory traffic is observed. Figure 8.16 summarizes all these impacts and shows the final results. Thus it is observed that the new cache design consistently performs better than the baseline configuration and in some cases, it can even outperform the higher associativity cache configuration.

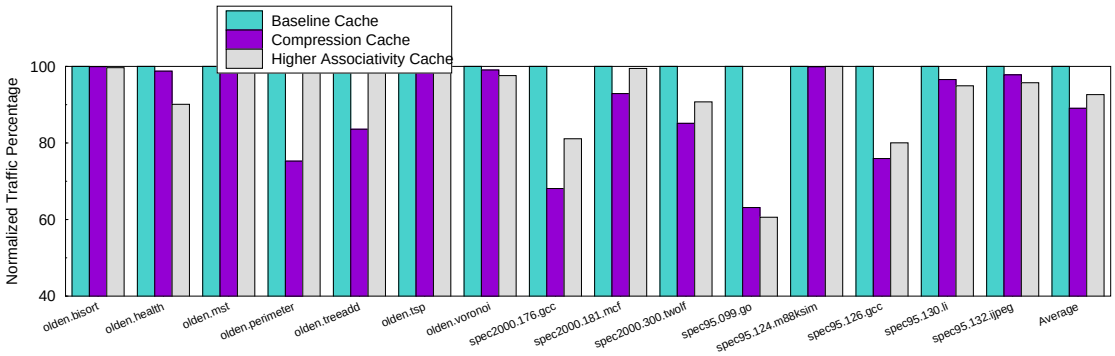


FIGURE 8.16. Comparison of memory traffic.

8.3 Related work

A number of different designs have been proposed to perform hardware and/or software prefetching to improve cache performance [49, 47]. Since prefetching might fetch unnecessary blocks or fetch blocks at the wrong time, it has the potential problem of wasting valuable memory bandwidth and polluting the cache. However, the new cache does not increase the memory traffic and a cache line is never replaced just to hold prefetched words. It also effectively transmits more words and reduces the memory traffic.

Some compression cache designs [33, 61] have been proposed to improve the data density inside the cache. In [33] a cache design is proposed that compresses two consecutive lines using a complex compression algorithm, both the compression and the decompression are expensive. As a result, it cannot meet the critical time constraints of a level one cache and is used at a lower level in the hierarchy. In [61] data is compressed using frequent values found from programs. If two conflicting cache lines can be compressed, both are stored within the cache; otherwise, only one of them is stored. Both of the above designs do not distinguish between the importance of different words within a cache line and a partially compressible cache lines cannot be exploited.

The pseudo associative cache [43] also has a primary cache line and a secondary

cache line. However, if a cache line enters its secondary place, it has to kick out the original line and hence there is a danger of converting a fast hit to a slow hit or even a cache miss. On the contrary, proposed compression cache design only stores a cache line to its secondary place if there are free spots. Neither will it pollute the cache line nor will it degrade the original cache performance.

8.4 Conclusion

A novel cache design is developed in this chapter to remove the value representation redundancy which was found in chapter 6. It partially prefetches the compressible words from the next cache line and stores these words in the cache. It removes the prefetch buffer and thus minimizes the cache size increase. Unlike other prefetching schemes that save the words in the cache, it never pollutes the cache line. On an average, the new cache improves the overall performance 7% over the baseline cache and 2% over the higher associativity cache configuration. The new design adopts the positive aspects of hardware prefetching and eliminates the problems it has, especially, it makes better use of both memory bandwidth and cache space. In this way, this design opens the way to apply hardware prefetching to more restricted environments such as embedded systems.

Chapter 9
Conclusion and future work

This dissertation makes contributions in the areas of program profiling and profile-guided compiler optimizations. While profile-guided optimizations can greatly improve program performance over the traditional ones, recent advances in profiling collect huge amount of profiling data and make information retrieval at analysis stage a bottleneck. On the other hand, due to the increasing performance gap between CPU and memory, new optimization opportunities arise from the fact that a significant percentage of the space stores redundant data. New type of profiles and profiling techniques are needed in the design of new optimization techniques. In this dissertation, these problems are solved through the design and application of new data compression techniques. In particular, the contributions are summarized in section 9.1. Section 9.2 discusses directions for further research.

9.1 Summary of contributions

TWPP+ representation. A new representation is proposed to compress whole program path profiles including both control flow and memory address information. While prior work aimed at compressing the profiles to achieve maximal compression ratio, the proposed timestamped whole program path representation puts more emphasis on organization and speed up the information retrieval in compiler analysis and optimization. Control flow and memory addresses are explicitly separated from each other. The complete control flow is represented by a two levels organization. A global call graph is kept to remember the calling context information. At the function level, a sequence of timestamps is attached to each basic block in the control flow graph to indicate when it is executed.

Memory address trace is explicitly represented as dependence edges and reorganized as a sequence of dependence edges attached at each load instruction point in the control flow graph.

Applications of TWPP+. Instead of considering a trace as a stream of symbols, TWPP+ divides a complete trace into a control flow trace part and a memory dependence trace part; each part is then reorganized to allow fast retrieval of information during data flow analyses. Common queries in data flow analyses could be processed much faster and thus could be used to integrate the execution information into a broad range of data flow analyses and optimizations. Three applications are demostrated in this dissertation to use the information contained in TWPP+ representation. It could be used to study the overall behavior of a program execution. By regrouping and sorting the memory dependence edges, redundant load and store instances are identified. A significant percentage of load instructions are highly redundant and could be further optimized to improve performance. With the timestamps, the exact execution order is maintained in the TWPP+ such that it is much faster to identify the frequency of some data flow facts at some program points with respect to the given whole program path. The TWPP+ representation can also be use as debug tool to create dynamic slices at any program execution point. Different slicing algorithms are simulated using this representation with different cost and slice accuracy tradeoff.

Type-based profiling for identifying value redundancy. A type-based profiling framework is proposed to profile the programs with respect to both high-level type information as well as value characteristics. It is implemented with a combination of instrumentation and simulation using SUIF compiler [54] and Simplescalar simulator [10]. In this framework, data types are profiled at field level; value range summaries are collected for each field. With this information,

candidate types for compression could be identified. A benefit-cost model based on profiles is used in the framework to assist the design and application of the new compression techniques.

Applications of value redundancy. Two new types of value representation redundancy are identified for small values and pointer addresses respectively. Those types of redundancy exist widely in a spectrum of programs. They are simple in logic and easy to explore in practice. Moreover, this dissertation proposed both software and hardware approaches to explore them.

Data compression transformations are proposed and implemented as a compiler approach. Code are transformed according to access pattern of the candidate data fields. To further reduce the runtime overhead, data compression instruction extensions are designed and evaluated. With the help of six new simple RISC- style instructions, the memory footprints are greatly reduced and the overall performance is improved on top of existing memory locality enhancement techniques.

A novel hardware cache design is proposed and evaluated to improve the program performance by reducing the number of cache misses. Compressible values are transmitted from the memory and stored in the cache in compressed formats. By removing value representation redundancy, the compression cache can effectively fetch and store more data items with the given memory bandwidth and the given cache size. The experiments further identified that in many programs, the prefetched compressed data items are more important for program execution. The overall performance is greatly improved from the compression with reduced cache misses and memory traffic.

9.2 Future work

Profile database. With the increasing program complexity, advances in program profiling tend to collect a huge amount of profiling data. This could be the result from collecting whole program paths, or from the iterative collections with different inputs. The former is discussed in this dissertation, the later has been employed in profiling complicated commercial programs. In all these cases, it would be helpful to design a unified interface for stored profiles. The profiles could be organized as an independent subsystem – a special database. Different analyses and optimizations could issue different queries to this subsystem and the queris are processed similar to that of SQL queries.

New optimizations. Compared to prior profiles, a whole program path provides accurate execution information. Especially, it keeps the information across the loop boundaries and procedural scopes. Reorganizing the whole program path at multiple semantic levels, TWPP+ can be used to enhance existing data flow analysis techniques as well as design new optimization passes. It would be very interesting to explore additional optimization opportunities using the information provided by TWPP+.

Dynamic slicing. Dynamic slicing is used as an example to illustrate the strength of the new timestamped whole program path representation. Although it is beyond the scope of this dissertation to fully explore dynamic slicing, it would be an interesting topic to evaluate different dynamic slicing algorithms with the presence of pointers and arrays in real C programs. The experience in our research group [63] showed that the memory requirement is extremely large if dynamically maintaining the data dependence edges. Since the accurate slicing algorithm could also be implemented by backward scan of the trace, it is more realistic to compare the implementations of the accurate slicing algorithm

using different representations and then choose the right representation to use in practice. While an accurate algorithm might execute longer, its memory requirement is well controlled.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, NY, 1990.

[2] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, UT, 2001.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[4] Thomas Ball and James Larus. Efficient path profiling. In *29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Paris, France, 1996.

[5] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 134–148, San Diego, CA, 1998.

[6] Ras Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–158, Las Vegas, NV, 1997.

[7] Ras Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, Montreal, Canada, 1998.

[8] Ras Bodik, Rajiv Gupta, and Mary Lou Soffa. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver B.C., Canada, 2000.

[9] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *The 27th International Symposium on Computer Architecture*, pages 83–94, 2000.

[10] Doug Burger and Todd Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, 1997.

[11] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 259–269, 1997.

[12] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, Vol. 1, 1999.

[13] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *The Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, CA, 1998.

[14] Martin C. Carlisle. *Olden: Parallelizing Progrms with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, 1996.

[15] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, Snowbird, UT, 2001.

[16] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, GA, 1999.

[17] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, 1999.

[18] Jack W. Davidson and Sanjay Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 186–195, 1994.

[19] Digital Equipment Corporation, Maynard, MA. *ATOM User Manual*, March 1994.

[20] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, 1998.

[21] Carole Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–32, 1998.

[22] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architecture and Compilation Techniques*, pages 102–115, San Francisco, CA, 1997.

[23] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *The 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558–568, Research Triangle Park, NC, 1997.

[24] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, pages 230–239, Chicago, IL, 1998.

[25] Richard E. Hank, Wen mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *IEEE/ACM International Symposium on Microarchitecture*, pages 158–168, Ann Arbor, MI, 1995.

[26] Doug Hunt. Advanced performance features of the 64-bit pa 8000. In *COMPCON'95*, 1995.

[27] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual*, 2002.

[28] Todd B. Knoblock and Erik Ruf. Data specialization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–225, 1996.

[29] Jens Knoop, Oliver Rthing, and Bernhard Steffen. Lazy code motion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–234, San Francisco, 1992.

[30] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[31] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver B.C, Canada, 2000.

[32] James Larus. Whole program paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, Atlanta, GA, 1999.

[33] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *IEEE International Conference on Computer Design*, pages 184–191, Austin, TX, 1999.

[34] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, pages 226–237, 1996.

[35] Chi-Keung Luk and Todd C. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *The 26th Annual International Symposium on Computer Architecture*, pages 88–99, Atlanta, GA, 1999.

[36] Eduard Mehofer and Bernhard Scholz. A novel probabilistic data flow framework. In *Lecture Notes in Computer Science 2027*, pages 37–51, Genova, Italy, 2001.

[37] MIPS Technologies Incorporation. *R10000 Microprocessor User's Manual-Version 1.1*, 1996.

[38] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[39] Robert Muth, Scott Watterson, and Saumya Debray. Code specialization based on value profiles. In *International Static Analysis Symposium, LNCS 1828*, pages 340–359, 2000.

[40] Craig Nevil-Manning and Ian H. Witten. Inferring lexical and grammatical structure from sequences. In *IEEE Data Compression Conference*, pages 179–188, Snowbird, UT, 1997.

[41] Soner Onder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, 1998.

[42] Vijay S. Pai, Parthasarathy Ranganathan, , and Sarita V. Adve. Rsim reference manual. version 1.0. Technical Report Technical Report 9705, Rice University, 1997.

[43] David A. Patterson and John L. Hennessy. *Computer Architecture, A Quantative Approach*. Morgan Kaufmann Publishers, 2nd edition, 1996.

[44] Alex Peleg and Uri Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):51–59, 1996.

[45] Ganesan Ramalingam. Data flow frequency analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 267–277, Philadelphia, PA, 1996.

[46] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural data flow analysis via graph reachability. In *The 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[47] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependance based prefetching for linked data structures. In *The Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, San Jose, CA, 1998.

[48] Vivek Sarkar. Determining average program execution times and their variance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, Portland, OR, 1989.

[49] Alan Jay Smith. Cache memories. *ACM Computing Survey*, 14:473–530, 1982.

[50] SPEC95. http://www.spec.org/osg/spec95/.

[51] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[52] Bernhard Steffen. Property-oriented expansion. In *International Static Analysis Symposium, LNCS 1145*, pages 22–41, Germany, 1996.

[53] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver B.C, Canada, 2000.

[54] Stanford SUIF. http://www.suif.org/.

[55] Trimaran. *The Trimaran Compiler Research Infrastructure*, November 1997. Tutorial Notes.

[56] Dan N. Truong, Franois Bodin, and André Seznec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329, Paris, France, 1998.

[57] Jon Tyler, Jeff Lent, Anh Mather, and Huy Van Nguyen. Altivec(tm): Bringing vector technology to the powerpc(tm) processor family. Phoenix, AZ, 2000.

[58] Scott Watterson and Samuya Debray. Goal-directed value profiling. In *International Conference on Compiler Construction, LNCS 2027, Springer Verlag*, Genova, Italy, 2001.

[59] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[60] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[61] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *IEEE/ACM International Symposium on Microarchitecture*, pages 258–265, Monterey, CA, 2000.

[62] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. Near-optimal intraprocedural branch alignment. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 183–193, 1997.

[63] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. *Dynamic Slicing Algorithms: Design and Evaluation*, 2002. Manuscript.

[64] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. In *ACM 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Cambridge, MA, 2000.

[65] Jacob Ziv and Abraham Lempel. A universal algorithm for data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[66] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.