Table of Contents

1 **Introduction** ... 1
- 1.1 Background ... 1
- 1.2 Addressing Range Considerations ... 3
- 1.3 Wide Bus Utilization ... 4
- 1.4 Cache Misses: Conflict and Compulsory ... 4
- 1.5 Interference in Unified Caches ... 6
- 1.6 Summary of Data Placement Opportunities ... 6
- 1.7 Outline of Ensuing Chapters ... 6

2 **Related Work** ... 8
- 2.1 Code Placement ... 8
- 2.2 Data Prefetching ... 12
- 2.3 Loop Transformations ... 13
- 2.4 Data Transformations ... 13
- 2.5 Load and Store Coalescing for Wide Buses ... 14
- 2.6 Reducing Range-Dependent Addressing Costs ... 15
- 2.7 Heap Optimizations ... 17
- 2.8 Cache Conscious Data Placement ... 18
- 2.9 The Potential for Improvement on Prior Work ... 20

3 **Data Affinity Analysis** ... 22
- 3.1 Code and Data Profiling in Prior Work ... 26
- 3.2 Interprocedural Static Data Analysis ... 29
  - 3.2.1 Intraprocedural Analysis ... 31
  - 3.2.2 Interprocedural Data Analysis ... 34
- 3.3 Inlining Functions in the CFGs of a Program ... 40
- 3.4 Path Analysis to Produce the Affinity Array ... 43
  - 3.4.1 Path Analysis without Complete Inlining ... 50
  - 3.4.2 Result of Path Analysis ... 52

4 **Data Placement** ... 54
- 4.1 Cache Line Packing ... 56
  - 4.1.1 The Greedy Algorithm for Cache Line Packing ... 59
  - 4.1.2 A Graph Matching Algorithm for Cache Line Packing 62
  - 4.1.3 Cache Line Packing by Graph Partitioning ... 63
  - 4.1.4 Other Methods for Cache Line Packing ... 66
- 4.2 Data Placement by Graph Partitioning ... 66
- 4.3 Fine Tuning the Graph Partitioning ... 70
- 4.4 Local Refinement of the Data Placement Solution ... 77
- 4.5 Padding Large Arrays to Minimize Conflicts ... 85
- 4.6 Data Placement for Local Variables ... 86
- 4.7 Reducing Bus Cycles with Load Coalescing ... 93
- 4.8 Reducing Addressing Costs for Global Variables ... 94
- 4.9 Reducing Code/Data Conflicts in Unified Caches ... 94
- 4.10 Code Generation after Data Placement ... 95

5 **Measurements and Results** ... 96
- 5.1 Benchmark Programs ... 96

5.2 Performance Results for Optimizations 102
5.2.1 Data Placement Of Local Variables 119
5.3 Analyzing the Optimization Stages 119
5.4 Comparison to Prior Work 127
5.5 Enhancements Suggested by the Measurements and Results 130
6 **Determining Cache Parameters and Retargeting the Optimizations** 131
6.1 Background 132
6.1.1 Dynamic Measurements Using Timing Tools 133
6.2 Robustness and Portability 134
6.2.1 Performance Counter APIs and Portability 134
6.2.2 Robustness and Reliability 135
6.3 Algorithms and Measurements 136
6.3.1 Data Cache Line Size 137
6.3.2 Data Cache Size 138
6.3.3 Data Cache Associativity 139
6.3.4 Write Allocation Policy 140
6.3.5 Replacement Policy 140
6.3.6 Data Cache Subblocking 141
6.3.7 Data Cache Miss Penalties 143
6.3.8 Data Cache: Split or Unified 145
6.3.9 Instruction Cache Line Size 146
6.3.10 Instruction Cache Size 146
6.3.11 Instruction Cache Associativity 147
6.3.12 TLB Measurements 147
6.4 Summary of Measurements 148
6.5 Implementation 149
6.6 Software Availability 150
6.7 Ongoing Work 150
6.8 Supplementing AMP with Microbenchmarks 150
6.8.1 Computing Benefits of Load and Store Coalescing.. 151
6.8.2 Computing Benefits of Immediate-Mode Addressing for Local Variables 152
6.8.3 Computing Benefits of Moving Cache Misses Earlier in Cache Lines 153
6.8.4 Computing the Cost of Cache Thrashing 153
6.9 Retargeting the Optimizations 153
7 **Implications for Hardware and Software Design** 156
7.1 Architecture Interactions 156
7.2 Implications for Microarchitecture Implementation 158
7.3 Implications for Software Engineering Practices 160
8 **Conclusions and Future Work** 162
8.1 Summary of Contributions of the Research 162
8.2 Future Enhancements and Extensions of the Research 164

TABLE 5-1: Benchmark programs. 100
TABLE 5-2: Cumulative sizes of global variables by category. 101
TABLE 5-3: Execution time speedup results. 103
TABLE 5-4: First-level data cache read misses. 104
TABLE 5-5: Second-level unified cache data read misses. 106
TABLE 5-6: Second-level unified cache data write misses. 107
TABLE 5-7: Second-level unified cache total data misses. 107
TABLE 5-8: Changes in instruction cache misses and invalidations. 110
TABLE 5-9: Data cache changes as a percentage of executed instructions. 111
TABLE 5-10: First-level data cache read miss rates. 113
TABLE 5-11: Effect of using split second-level caches. 114
TABLE 5-12: Associativity effect on DYFESM data cache misses. 116
TABLE 5-13: First-level cache write misses. 118
TABLE 5-14: Data TLB misses. 118
TABLE 5-15: Path analysis path terminations by cause. 120
TABLE 5-16: Cache line packing data. 121
TABLE 5-17: Relative conflict reductions by optimization stage. 122
TABLE 5-18: Relative benefits of the best five swaps and all rotations. 123
TABLE 5-19: Compilation times in seconds. 125
TABLE 5-20: Compilation time in seconds by stage. 126
TABLE 5-21: Compilation time in seconds by stage (continued). 127
TABLE 5-22: Comparison with CCDP (different benchmarks used). 129
TABLE 6-1: Measured Cache and TLB Parameters 148
TABLE 6-2: Miss Penalties for Various Intel CPU Personal Computers and SGI MIPS .. 149

Temporal relationship graph. 11
Pruning CFGs. 37
Further CFG pruning examples. 38

Chapter 1

Introduction

Many recent research efforts in the field of optimizing compilers have focused on the placement of data. Careful placement of data items has been demonstrated to reduce cache misses and execution time in popular benchmark programs. These efforts have been concentrated on just a few types of data (large arrays, and heap-allocated objects) and in only a few application program categories (scientific array-processing programs, and programs that primarily traverse pointers across recursive data structures). The first level data cache has been the primary focus of most of these efforts. It appears that these research areas are nearing maturity, as will be seen in the next chapter.

However, this research will demonstrate that data placement optimizations can have greater scope and impact than previously thought. Several architectural features of modern microprocessors interact with the placement of data and affect performance. Specifically, (1) limited addressing ranges and use of registers for addressing, (2) wide data buses, and (3) multilevel cache hierarchies, each require data placement optimizations to ensure their efficient use.

After providing background information about these problems in this chapter, we will discuss related work in the next chapter. We then devote the remaining chapters to exposition of a unified data placement framework that efficiently utilizes all of these architectural features while still being easily retargeted to numerous machines with different microarchitectures and memory hierarchies.

1.1 Background

The processor and various memory hierarchy components of a modern computer system are constructed from a variety of technologies that are improving at differing rates. Processor speed is primarily determined by on-chip transmission and transistor switching speeds, which are improving rapidly as semiconductor processes evolve. Cache memories

are constructed from SRAM chips that also depend on transistor switching and use the same semiconductor processes as CPUs. However, at some point in a modern multi-level cache system, physical boundaries must be crossed that do not depend primarily upon transistor switching. Driving current through chip pins and printed circuit board traces, charging and discharging capacitors in a DRAM chip, and accessing electromechanical mass storage devices are all examples of memory hierarchy technologies that are not driven primarily by transistor switching speeds. None of these elements of the computer system can be expected to keep up with the speed improvements of CPUs; e.g., the 50% annual improvement in processor speeds has been widely contrasted with about a 7% annual improvement in DRAM first-access times ([Hennessy and Patterson 1990], p. 426).

Both hardware and software approaches that address this problem have been attempted. In hardware, the addition of pipelined memories and cache memories, including as many as three levels of cache memory in some systems, has been refined over several decades [Boland et al. 1967, Chen and Baer 1992, Jouppi and Wilton 1994, Wilson and Olukotun 1997, Bodin and Seznec 1995]. Recent architectural advances attempt to use the instruction-level parallelism (ILP) of a program to efficiently utilize otherwise wasted CPU cycles while waiting on the memory hierarchy [Jourdan et al. 1995, Hara et al. 1996, Morano et al. 2003]. Hardware cache controllers, some assisted by compiler-generated hints, have been designed to reduce cache misses and miss penalties [McKee et al. 1996, Johnson 1998].

A few software approaches to reducing memory hierarchy delays have also been tried. Data prefetching attempts to mask cache miss penalties by issuing non-blocking data fetches that are scheduled many CPU cycles before the value is used to give the cache time to load the new line [Mowry et al. 1992, Luk and Mowry 1996, Annavaram et al. 2001, Solihin et al. 2002]. Various loop optimizations in compilers have attempted to optimize memory access patterns by reordering computations, along with such techniques as internal array padding to reduce intra-array conflicts. This area of compiler research is fairly mature [Abu-Sufah 1979, Kandemir 2001, Irigoin and Triolet 1988, Wolf and Lam 1991, Carr et al. 1994, Chatterjee et al. 2001]. More recently, data cache miss reduction in code that intensively uses pointers and heap data objects has been achieved via reordering within pointer-linked data structures [Kistler and Franz 2000] and also by passing cache placement hints

to custom heap allocators [Calder et al. 1998, Chilimbi and Larus 1998] and by redesigning allocators to optimize cache locality and not just to reduce memory fragmentation [Grunwald et al. 1993, Shuf et al. 2002]. These software research efforts will be surveyed in more detail in the next chapter.

In addition to the delays at each level of the memory hierarchy, other aspects of the architecture of modern computers may cause a degradation of performance if an application has poor data locality. The sections below will examine two of these considerations, namely the expense of addressing data objects in memory and the efficiency of utilization of the buses that connect each level of the memory hierarchy to adjacent levels.

1.2 Addressing Range Considerations

Most processor architectures have several available addressing modes in the instruction set. For example, the address of a data item could be produced by specifying two registers whose values are added (register plus register addressing), or an instruction could encode a base register and an immediate value offset to be added to the base register (base register plus offset addressing). Base register plus offset addressing is generally more efficient for addressing local data, as the base register is the frame pointer register which is set upon entry to a function and remains invariant throughout the function, while the immediate offset is computed at compilation time when the compiler arranges the local variables in the stack frame. Thus, both the base register and the immediate offset are available without further computation at the time that a local variable needs to be accessed. However, if a stack frame is large enough to exceed the range of the immediate offset, then the base address of the data item will have to be computed in a register other than the frame pointer when the data item is used in an expression, which will be more expensive than using the invariant frame pointer register as a base address. The dynamic frequency of this more expensive addressing of locals can be controlled by data placement within large stack frames.

Similarly, global variables are most often addressed by computing the base address of the global in a register just prior to its use, then computing offsets either in an immediate field (if the offsets can be computed at compilation time) or in another register (as is the

case when looping through a data array using a loop index as the offset). In either case, the base register computations are not generally available in the header of each function, unlike the setting of the frame pointer, making accesses to global variables more expensive than accesses to local variables. If global data items were grouped by a data placement optimization according to their temporal locality in a program, then several global items accessed within a function could fall within the range of an immediate offset. By computing a global base address for these several variables into a register just once, in the function header, the address computations on most architectures could be reduced to a single load instruction at the point where each global is accessed. The size of the immediate addressing range differs from one machine to another, so this optimization is machine dependent.

1.3 Wide Bus Utilization

A typical first-level data cache memory fetches 16, 32, or even 64 bytes from lower levels of the cache and memory hierarchy to satisfy a cache miss. The data item requested by the CPU is then provided over the cache-to-CPU-registers bus. Because many CPU designs now include 64-bit registers, it is common to have a 64-bit (8-byte) cache-to-CPU bus today.

However, many application program variables are still declared as 32-bit objects, or even as 16-bit or 8-bit data types. Thus, a portion of the 64-bit bus is idle when transferring a 32-bit value from the cache into a CPU register. If two consecutive 32-bit memory addresses were requested by load instructions, these loads could be combined, or *coalesced*, into a single 64-bit load that only requires one bus transaction rather than two [Davidson and Jinturkar 1994]. Rather than combine loads that happen to be from consecutive addresses, a compiler could perform a data placement optimization that maximizes the number of loads (or stores) that can be coalesced.

1.4 Cache Misses: Conflict and Compulsory

A request for a data item not found in the cache is a *cache miss*. Cache misses are generally classified into three categories: *Compulsory* misses, so named because they are caused by referencing a data item for the first time in a program's execution history; *capacity* misses,

caused by referencing more cache lines over time than can fit into the cache; and *conflict* misses, caused by referencing different memory items which map to the same cache line number in the cache and exceed its ability to hold them all by its associativity (and which are neither compulsory nor capacity misses). For example, with a direct-mapped 16KB data cache, when a 32KB array is completely and repetitively processed from start to finish, the first 32KB set of references will generate compulsory misses, while the next complete processing of the array would generate capacity misses, as a 16KB array cannot be expected to hold 32KB of data. By comparison, if a program accessed repeatedly two array elements at offsets 0 and 16KB from the start of the array, the first two accesses would generate compulsory misses, while further misses would be classified as conflict misses, because these two addresses within the array will map to the same cache line number in the direct-mapped cache. Even though the program only accesses a few bytes within the array, and thus capacity is not a problem, the cache will be said to *thrash* as we repeatedly alternate accesses between these two elements. If, instead of a single array, a program had two consecutive 16KB arrays and accessed offset 0 within each array, this conflict miss thrashing could be avoided by placing the arrays in non-consecutive fashion, perhaps using some smaller variables between them, so that the starting offsets of the arrays did not map to the same cache line. This suggests that a data placement optimization can reduce conflict misses in a cache.

Furthermore, if variables within a program are not large enough to fill a cache line, they will share that cache line with other variables. When the first reference to any of these variables occurs, the compulsory miss will cause the entire cache line to be filled. References to other variables in that same line will then not suffer misses, due to the property of *spatial locality* exhibited, namely that references to items at addresses close to each other have occurred. A data placement optimization can reduce compulsory misses by grouping small variables together in a cache line if they are referenced close enough together in the execution time of the program to have the cache line still be present when the second reference occurs. It should be noted that different cache levels in a cache hierarchy often have different line sizes, with the bigger and lower levels having larger lines. Where this is the case, the cache line packing optimization must be designed to reduce compulsory misses at more than one level of cache by using a hierarchical packing algorithm.

1.5 Interference in Unified Caches

The most common design for a cache hierarchy is to have separate, or *split*, first level instruction and data caches, with second and lower level caches *unified*, i.e. containing both instructions and data. The split caches at the first level of most cache hierarchies offer the advantage of higher bandwidth, as instruction fetches and data fetches can be in progress simultaneously from separate caches over separate busses. Unified caches at the second and third cache levels have been observed to dynamically adapt themselves well to different applications, some of which reference a larger data set than their instruction stream, and some of which are the opposite. However, unified caches create the potential for conflicts between code and data, such that a data item is evicted from a unified cache by an instruction cache miss, causing a later reference to that data item to be a conflict miss. A data placement optimization should place data at addresses that do not conflict with the instruction stream that references that very same data set in order to be effective for multilevel cache hierarchies.

1.6 Summary of Data Placement Opportunities

It is apparent that a comprehensive data placement optimization would enhance program performance if it could simultaneously reduce addressing costs, reduce bus cycles, reduce conflict misses between data items, reduce compulsory data cache misses, and reduce code/data conflict misses in unified levels of the cache hierarchy. It is shown in the next chapter that prior work has addressed some of these concerns but not others, and no data placement optimization has addressed all of these performance issues in concert.

1.7 Outline of Ensuing Chapters

The remainder of this dissertation is organized as follows. Chapter 2 examines related prior work and the potential for improving upon those efforts. Chapter 3 presents the data analysis framework that provides the necessary information for data placement to proceed. Chapter 4 presents the data placement algorithms in detail, explaining the many design decisions and alternatives. Chapter 5 contains the measured results of applying our optimizations on various benchmarks. Chapter 6 addresses retargeting issues, the machine-

dependent information needed to implement the proposed data placement optimizations, and how to obtain that information. Chapter 7 examines the interactions between our optimizations and architectural design decisions, including lessons that computer architects could apply in their future designs to maximize performance and reduce hardware complexity in the presence of any compiler that implements our optimizations. Interactions with software engineering methodology are presented in this chapter as well. Chapter 8 presents a summary and conclusion.

Chapter 2

Related Work

Prior work has addressed memory hierarchy related optimization in several ways. Code placement reduces conflict (and sometimes capacity) misses in the instruction cache. Prefetching attempts to reduce data cache misses without directly addressing placement issues. Loop transformations reduce data cache misses for applications that spend most of their time processing large arrays. Data transformations change the layout and size of arrays to reduce intra-array conflicts. Heap data placement reduces data cache conflicts between heap objects. Cache conscious data placement reduces conflicts in the first level data cache by placing global and heap variables. Each of these areas of prior work will be discussed in the following sections, along with their limitations and the need for further work.

2.1 Code Placement

Code placement work has been published at least since 1989, beginning with McFarling's work that used profiling information to reduce direct-mapped instruction cache misses by combining code placement with cache bypassing [McFarling 1989]. Each basic block was placed in whatever order was deemed best by the placement algorithm, not being confined to the original ordering of the program. Certain instructions that created cache conflicts, even after code placement, were tagged as ineligible for caching, because the cost of caching exceeded the benefit. A hypothetical architecture that could make a caching decision on each instruction (with each instruction cache line being only one word long) could make use of this optimization. Results obtained by simulating such an architecture demonstrated cache miss reductions of as much as 61% (using cache bypassing) or 19% (without cache bypassing). Reducing the instruction cache misses by 19% translated into a 1.7 percentage point increase in the (already high) hit rate of the cache. No timings were made on real machines.

Pettis and Hansen targeted not only cache misses but expensive branch instructions [Pettis and Hansen 1990]. On their Hewlett-Packard architecture, branches beyond a certain range required a more expensive instruction sequence than branches within that range. Accordingly, they used profiling information to group basic blocks into popular and unpopular sets, with the popular blocks placed closer to the top and unpopular blocks closer to the bottom of the procedure. A third set of basic blocks, called *fluff*, were never observed to execute at all during profiling (perhaps they contained error handling code, for example). These fluff blocks could be physically separated from the rest of the procedure's blocks and grouped into a memory region with fluff blocks from other procedures, a technique they called *procedure splitting*. A greedy algorithm then built a chain of procedures beginning with the two procedures that had the most calls and returns between them, adding to the chain until all procedures were ordered in the chain. This ordering reduced cache conflicts, as adjacent placement would minimize the overlap between two procedures in the cache. This combined approach of making often-executed blocks adjacent within a procedure while making procedures with mutual affinity in the call graph adjacent in the final code placement by the linker reduced the expensive branches while reducing instruction cache misses at the same time. Pettis and Hansen demonstrated that placing procedures alone was not as effective as reordering basic blocks within procedures first and then placing the procedures. Procedure placement achieved an 8% reduction in run time on their system, basic block reordering a 12% reduction, and the combination a 15% reduction. All figures are due partly to cache miss reduction and partly to reduction of the execution count of the long branches.

Hashemi, Kaeli and Calder modified the approach of Pettis and Hansen [Hashemi et al. 1997]. They labeled the procedures of the program as popular or unpopular depending not on their execution time, but on the frequency of calls into and out of each procedure. They then created a map of the cache, based on the actual cache size and cache line size, and began building a chain of procedures, starting with the most popular edge between a pair of procedures in the call graph, as in Pettis and Hansen. After placing a pair of procedures, the cache lines used are marked with arbitrary *colors* and further procedures are considered for adding to the chain. If the next most popular edge connects two procedures, neither of which was in the first pair of procedures just placed, then these form a new pair

with arbitrary colors assigned. If the next most popular edge connects a procedure from one placed pair to a procedure in the other placed pair, then the possible orderings (e.g. for procedure pairs A-B and C-D, the possible chainings are A-B-C-D, B-A-C-D, A-B-D-C, and B-A-D-C) to place the two procedures with the popular edge between them close together (e.g. given A-B and C-D, with new popular edge B-D to process, an order A-B-D-C would be chosen). Then, if this ordering causes cache line conflicts between B and D, a gap will be inserted between B and D if such a gap eliminates or reduces the conflicts. If so, a best-fit search of the unpopular procedure list will attempt to find a way to fill this gap. If necessary, there will be gaps in the code layout. Thus, in contrast to Pettis and Hansen, the minimization of cache conflicts is directly addressed via cache line coloring while the procedure chain is being built. Measured results are a small improvement on Pettis and Hansen's results on about half of all benchmarks, and statistically identical on the other half.

Following up on this work, Gloy, Blackwell, Smith, and Calder apply the same basic approach (chaining popular procedures, filling in gaps with unpopular procedures) with a few differences [Gloy et al. 1997]. The most important innovation is the profiling method, which produces a finer-grained data structure called a *temporal relationship graph* (TRG). The weighted call graph (WCG) of a hypothetical program (mimicking an example from their paper) is shown in Figure 2-1(a). From the WCG, we can determine that the main procedure, A, calls procedures B and C 40 times each, and it calls procedure D 20 times. Unfortunately, this tells us little about the cache behavior of the code. The dynamic procedure call trace of the program could be: $\texttt{A(BCBCD)}^{20}$, meaning that procedure A goes through a loop 20 times, calling B, then C, then B, then C again, and then D. The same weighted call graph could correspond to the procedure call trace $\texttt{A((B)}^{4}\texttt{D)}^{10}\texttt{((C)}^{4}\texttt{D)}^{10}$, meaning that procedure A goes through a loop 10 times, calling B 4 times and then calling D once on each iteration, and then goes through a similar loop 10 times calling C and D. In the first case, if procedures B and C map to the same cache lines, they will thrash (cause excessive alternating conflict misses), while in the second case, procedures B and C could be mapped to the same cache lines with no conflict misses at all. Clearly, the WCG is not sufficient to distinguish between these two situations, which have very different implications for code placement.

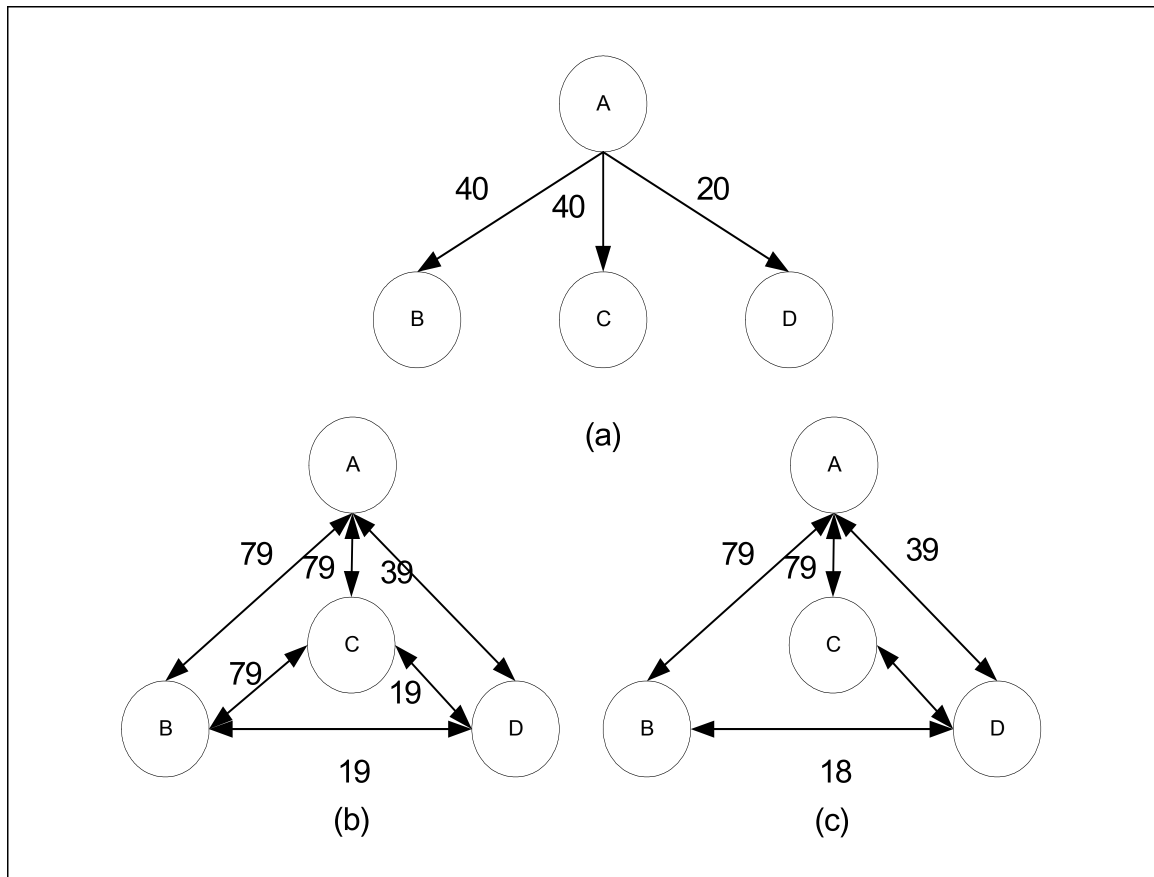


Figure 2-1. Temporal relationship graph.

To address this shortcoming, Gloy et al. developed the TRG. At all procedure call and return points, a call is made to a profiling procedure that maintains an ordered set of recent procedure references. If the current call or return is to procedure B, for example, the profiling procedure scans backwards in the set to find another instance of B. If found, then all of the references to other procedures in between the old reference to B and the current reference to B are recorded in the TRG, with new nodes and/or edges added as needed. Upon completion of profiling, the TRG accurately represents the potential conflicts in the instruction cache between pairs of procedures. For example, the second hypothetical dynamic procedure call trace above will become the call-and-return trace: $\texttt{A((BA)}^{4}\texttt{DA)}^{10}\texttt{((CA)}^{4}\texttt{DA)}^{10}$ (because procedure returns are now represented in the trace, not just calls, as a return to a calling procedure causes the instruction fetches to be made from the procedure to which control is returned, affecting the instruction cache, and hence returns are modeled explicitly.) The TRG produced is shown in Figure 2-1(c). The first procedure call trace above corresponds to the TRG in Figure 2-1(b). Notice that the

different temporal relationships between procedures B and C are now accurately modeled. In Figure 2-1(b), we can clearly see that B and C would conflict in the cache, while Figure 2-1(c) clearly shows that they do not have overlapping lifetimes and will not conflict in the cache. Unfortunately, the TRG profiling method causes the profiling run to take approximately 25 times as long as a normal run of the program. Thus, it is not practical for use on long-running programs, which are prime candidates for high levels of compiler optimization.

Further code placement work was done in the dissertation of Kalamatianos using yet another profiling method, which produced a Conflict Miss Graph (CMG) [Kalamatianos 2000]. The CMG uses a stack of procedure references, similar to the TRG approach, to model the interaction between procedures in the cache. The CMG operates at the cache line level and counts the worst case number of cache line conflicts that each procedure can cause another procedure to suffer. Code placement was done both with a first-level cache model, and with a multilevel cache model. Caches of arbitrary degrees of associativity could be modeled, whereas Gloy et al. targeted direct-mapped caches and described how the work could be extended to higher associativity caches. Otherwise, the same cache line coloring placement approach is used in both authors' work. Measured results were almost identical between the CMG and TRG approaches. Altering the placement algorithm to take into account more than one level of instruction caches did not lead to improvement. This was partly due to the fact that the placement algorithms were unable to address both L2 cache misses and page locality (and hence, instruction TLB (ITLB) misses). One of these metrics was always sacrificed to the other in the various algorithm versions tried. Another reason that multilevel placement did not yield benefits is that the L2 cache is shared, and conflicts between data and code must be modeled and reduced in order to optimize the performance of this level of the cache. No data placement was undertaken in this work.

2.2 Data Prefetching

Data prefetching attempts to mask cache miss penalties by issuing non-blocking data fetches that are scheduled many CPU cycles before the value is used to give the cache time to load the new line [Mowry et al. 1992, Luk and Mowry 1996]. If the prefetched line

displaces a cache line that will be needed even before the prefetched line, then the prefetch was a mistake and degrades performance. If the prefetch requests a line that is already in the cache, then the prefetch instruction was an unnecessary bloat in the instruction stream and degrades performance. Thus, prefetching is best implemented within a data placement framework, where it can be used only for the few data variables that are known to be placed unavoidably in conflict with other objects that will have evicted these variables from the cache, making a prefetch profitable. Furthermore, the data placement framework will be able to determine "how far is too far" when hoisting the prefetch up to a location before the data is needed, because it will know when the hoisting of the prefetching load is crossing over another load that conflicts with the prefetched variable. Prefetching could be quite valuable within such a placement framework, but it is certainly not a substitute for data placement.

2.3 Loop Transformations

Various loop optimizations in compilers (loop tiling, loop inversion, cache blocking, etc.) have attempted to optimize memory access patterns by reordering computations [Irigoin and Triolet 1988, Wolf and Lam 1991, Carr et al. 1994, Abu-Sufah 1979]. One of the most precise efforts in this regard has been based on cache miss equations, which model the cache misses that will be produced by a given loop nest if the array subscript expressions of the arrays accessed within the loop nest are analyzable at compilation time, and apply loop transformations to directly address the cache conflict problems [Ghosh et al. 1999]. Because of the importance of fine tuning the performance of scientific applications that heavily process large arrays in loop nests, this area of research is probably the most mature within the general area of data cache miss reduction.

2.4 Data Transformations

Bacon et al. developed a compiler framework that analyzed affine array subscript equations and determined where there would be intra-array and inter-array conflicts [Bacon et al. 1994]. To solve intra-array conflicts, they introduced padding of dimensions within the array, e.g. making each column in a Fortran array longer by adding rows. To solve inter-

array conflicts, they calculated optimal inter-array padding distances, then reordered arrays and added dead space between arrays where it was needed. These kinds of data transformations, as with all others we are aware of, are applicable only to programs that spend the bulk of their time in array processing.

2.5 Load and Store Coalescing for Wide Buses

Davidson and Jinturkar examined the effectiveness of coalescing array loads and stores on machines with buses wider than some common data types, such as single-precision floating-point values [Davidson and Jinturkar 1994]. Their implementation unrolled loops in order to produce more than one load or store per loop iteration, then coalesced the loads or stores in order to reduce the number of loads and stores performed and hence reduce the number of bus cycles. After each coalesced load, the results must be unpacked into 2 or more destination registers. Similarly, before each store, results must be packed into a single register. Because most architectures have more efficient instruction sequences for unpacking bit fields from a register than for packing bit fields into a register, coalescing stores was not as productive as coalescing loads. In fact, coalescing stores was not beneficial on any of the three architectures tested (the DEC Alpha 21064, Motorola 88100, and Motorola 68030), and the expense of the unpacking instructions caused the benefits of load coalescing to vary across the architectures, even dropping into the negative range for the Motorola 68030, while DEC Alpha loop kernels saw execution time reductions of up to 40%. Operating on array items within unrolled loops, no data placement was involved in this optimization.

Nandivada and Palsberg performed a load and store coalescing optimization for the Intel IXP1200 network processor that relied on data placement [Nandivada and Palsberg 2003]. This architecture is unusual in that it supports loading an eight-byte data value into two separate four-byte registers in a single load instruction, which is ideal for the load coalescing optimization. Furthermore, the eight-byte data item could be aligned on a four-byte boundary, where most architectures would require eight-byte address alignment for an eight-byte load. With these advantages, they formulated the placement problem as an integer linear program (ILP) and used a commercial ILP solver as a library routine to determine

a near-optimal placement of scalars. The execution time reductions ranged from 0.8% to 15.1% on the benchmarks tested. However, they make no mention of cache effects or cache measurements, so it is not clear how to separate the cache effects of their placement optimization from the load coalescing effects. The optimization was not integrated with a cache-conscious data placement optimization. In addition, their peephole optimization phase coalesces non-scalar loads and stores where this can be done without loop unrolling, and no measurements are given to separate the benefits of these non-placement-related coalescings from the benefits derived from the placement-related coalescings.

2.6 Reducing Range-Dependent Addressing Costs

No prior work in reducing addressing costs of global variables for general purpose processors through data placement existed when this research began. Srivastava and Wall performed a link-time optimization for the Alpha AXP processor that reduced accesses to the Global Address Table (GAT) that was used for accessing all global variables on that machine [Srivastava and Wall 1994]. As on many machines, the Alpha collected the addresses of all globals accessed in a function into a GAT for that function. In order to load the value of a global variable, two memory accesses were required. The first access loaded the address of the global variable from its entry in the GAT, and then the second memory access used that address to load the value of the global variable. Separate GATs were created for each function because of a lack of knowledge at compile time concerning program size, distances between function, separate compilation units, dynamic and shared libraries, etc. At link time, however, the entire program is visible, and GATs for adjacent functions can be merged, the global pointer register can be made to point to a merged GAT and thus does not have to be saved and restored around function calls, etc. Some variables might be within immediate offset range of the global pointer register; thus, instead of using the global pointer register to load the address of the global from the GAT, the value of the global could be loaded in one step. No rearranging of data was performed in this optimization, although the linker changed code sequences. An average improvement of just more than 3 percent was seen on the benchmarks tested.

As this dissertation was being completed, two new papers have been published on reducing addressing costs for global variables by means of link-time [Haber et al. 2003] and post-link-time [Luk et al. 2004] optimizations. The link-time work of Haber et al. placed global data in order of its frequency of use and positioned the IBM PowerPC GAT just before the most frequently accessed data item. This permits some two-step accesses to the most frequently accessed variables to be made in one step by using an immediate offset from the global pointer register. Furthermore, the grouping by frequency of access makes an indirect improvement on data locality and hence should reduce cache misses. A profiling run is made first, then the link-time optimization uses the profiling data as feedback. Because the data reordering is not aimed specifically at reducing cache misses, but only at increasing opportunities to optimize the instruction sequences, the data reordering itself had a negative effect on five of the ten benchmarks, a negligible effect on two others, and a positive effect on the other three benchmarks. The reductions in accesses to the GAT overcame these problems and produced an average improvement in execution time of three percent.

The post-link-time optimizations of Luk et al. on the Itanium processor included a large set of diverse optimizations, making it hard to separate the benefits of each from the rest. These optimizations included code placement, data placement, code prefetching, data prefetching, GAT addressing reduction, function inlining, dead-code elimination, branch forwarding, and store-load forwarding. The first four (code and data placement and prefetching) were considered the most significant by the authors. In one pathological case, a hot spot for cache thrashing in consecutive accesses to heap-allocated variables was corrected by inserting prefetch instructions into the *SPECint2000* benchmark *181.mcf*. This reduced execution time by forty percent. Even with this improvement, the geometric mean across the benchmarks tested was about eight percent, with most benchmarks in the two to four percent improvement range. This research is notable for its use of performance counters to gain a statistical profile, which involves much less overhead than the typical dynamic profiling by code instrumentation. The data placement was done to reduce cache misses, and the GAT access reductions were then obtained from the final data layout. The data placement was shown to increase the opportunities for GAT addressing optimizations, but the placement was not tuned specifically for that purpose.

2.7 Heap Optimizations

Grunwald, Zorn and Henderson demonstrated the cache and paging inefficiencies of memory allocators that concern themselves only with space efficiency [Grunwald et al. 1993]. More recent work has measured the improvements in various locality-sensitive allocation algorithms [Chilimbi and Larus 1998, Boehm 2000].

Kistler and Franz observed that many operations on recursive data structures that reside in the heap, such as linked lists, involve nothing more than accessing the key field of a list item, comparing it to some value, and then following a pointer to the next item in the list [Kistler and Franz 2000]. Given that fact, it would be very advantageous to have the key field and the pointer to the next element in the list reside in the same cache line, regardless of how the data structure was defined by the programmer. Furthermore, they claimed that it is faster to access the words within a cache line approximately in order, avoiding the pathological case in which the *n*-th word is accessed, and then immediately the (*n-1*)-st word is accessed, on a cache miss. In this case, the load-forwarding hardware will return the cache line from memory in the wrapped around order that starts with the *n*-th word and concludes with the (*n-1*)-st word, and the request for the (*n-1*)-st word will have to wait for the cache line fill operation to complete, creating stall cycles. Using a dynamic path profiling and dynamic (i.e. run-time) optimizer for the language Oberon-2, they profiled the fields within structures (i.e. heterogeneous data objects) and created a temporal relationship graph (TRG) for each structure. Using the TRG, recursive graph bipartitioning (i.e. bisecting) is used to divide each structure into smaller and smaller clusters of data fields until the size of a cache line is reached. Fields are arranged within cache lines to avoid penalties in cache-line-fill hardware, as well as to reduce memory bank conflicts. No measurements of the effectiveness of these rearrangings within the cache lines are provided. Six benchmarks that intensively allocate heap objects are optimized and measured, with execution speedups ranging from 1.03, 1.04, and 1.15 at the low end, to 1.30, 1.69 and 1.96 (i.e. almost a 50% reduction in execution time) at the high end. When considering the runtime cost of continual path profiling and optimization, the numbers worsened only slightly, with one speedup dropping into the negative range and the others coming close to their original improve-

ments. Work similar to this research was also performed by Chilimbi and Larus et al. [Chilimbi et al. 1999a, Chilimbi et al. 1999b].

2.8 Cache Conscious Data Placement

Panda, Dutt, and Nicolau examined the placement of global scalar and array variables to reduce cache misses in the CW4001 embedded processor from LSI Logic [Panda et al. 1997]. The data cache for typical configurations of this embedded CPU core are only 256 bytes, and extremely small and simple benchmarks were used. A simple greedy algorithm is used to cluster scalars together into cache lines, and then all cache-line-sized clusters are placed relative to each other to minimize cache conflicts. Finally, arrays are placed with a greedy method that uses gaps between arrays as needed to minimize conflicts. Nothing is said about placing the clusters of scalar variables in order to minimize conflicts with the arrays, or vice versa. The embedded processor used has no TLBs or virtual memory paging behavior to be addressed. Both a direct-mapped and a two-way associative data cache are measured. The benchmarks measured were tight loops and kernels with single-digit numbers of array variables and a similar number of scalar variables. Execution time speedups ranged from zero to almost a 50% time reduction for the direct-mapped 256 byte data cache, and from zero to almost a 40% reduction for the two-way associative data cache. Average speedup factors were 1.34 and 1.21 for these two caches. All measurements were from a simulator of the embedded CPU running on a workstation.

Calder, Krintz, John, and Austin published the most comprehensive approach to data placement to date [Calder et al. 1998]. Their work was a successor to a portion of Austin's dissertation [Austin 1996]. In that dissertation research, Austin modified executable code to create unique global names for all stack, global, and heap variables, profiled these variables during execution, and then used straightforward greedy algorithms to place the global data objects relative to each other to reduce cache conflicts. Then, the entire stack frame was treated as a single large object (no rearranging of data within stack frames was done), and placed at a starting offset to minimize conflicts between stack and global data. Finally, heap objects, identified by the line number of their creation and the procedure stack backtrace at the time of their creation (to model different paths to the same allocation

points), were placed to reduce intra-heap conflicts. This placement was accomplished with a custom heap allocator that would accept cache offset hints inserted by the compiler. This work was extended by Calder et al. to use a TRG (temporal relationship graph) as the profiling data structure, and to place the stack relative to the global constants only, while placing the global and heap variables in one final integrated step. Small globals were grouped together on the basis of temporal locality to create spatial locality within cache lines. A 23.75% reduction in data cache misses (in an 8KB direct mapped first level cache) was measured, with 70% of the reduction coming within global variables and 25% from stack variables while only a little over 3% came from heap variables and 1% from global constants. Page locality worsened slightly. Execution time results were not given, although it was mentioned that the custom heap allocator that must take cache hints into account at run time might not be worth the price that is paid, given the small reduction in heap object cache misses on most of the benchmarks used. Increased spatial locality within cache lines was not measured.

Petrank and Rawitz recently proved that cache conscious data placement is in "the small family of extremely inapproximable optimization problems" [Petrank and Rawitz 2002]. That is, "if $P \neq NP$, then one cannot efficiently approximate the optimal solution even up to a very liberal approximation ratio." They demonstrate that, even if all profiling and accompanying data structure problems were solved to perfection, so that the placement algorithm had a complete ordered list of the memory references that will be made, a polynomial time placement algorithm still cannot approximate the optimal placement within any positive ε. Furthermore, they demonstrate that, given the kind of profiling information usually available (i.e. pairwise conflict metrics between pairs of data items), not even an exponential time algorithm can find a solution that is within an factor of *k-3* of the optimal cache miss reduction, where *k* is the number of blocks in the cache. Thus, the best that can be hoped for is to find good heuristics that will work well for a large number of benchmark programs.

2.9 The Potential for Improvement on Prior Work

Having reviewed the relevant work, we can find the motivation for new research that improves upon these efforts in several ways:

- The fact that the memory-to-cache bus is an element of the memory hierarchy to be used more efficiently has received little or no attention in most data placement work.
- Placing data to minimize use of the expensive addressing operations that are needed for data located far from the value held in an address register has only been addressed at link time or later, and only as a by-product of data placement, or even in the absence of data placement. Data placement has not been designed during compilation to maximize the effectiveness of the later address cost minimization. Architectures such as the Sun UltraSparc family [Weaver and Germond 2000], which does not use a global address table, will not benefit from past research efforts without considerable tailoring of the optimizations.
- The best code and data placement efforts have relied upon profiling runs that take an impractically long time to run, discouraging adoption of this research into commercial compilers.
- Many published results have targeted very small caches. The benefits of various cache miss reduction efforts have been known to lessen dramatically as cache sizes increase, calling into question just how useful some of the prior optimizations are for modern cache hierarchies. Measurements on first-level caches larger than 8KB are needed, along with measurements of second-level cache misses.
- Prior data placement research has targeted direct-mapped caches, with descriptions of (often complicated) extensions that could be implemented to target caches with higher associativity.
- Minimizing conflicts between code and data in unified levels of cache has been talked about as "future work" for years but never studied, or even measured for its potential impact throughout the cache hierarchy.
- Prior data placement algorithms have been fairly simple greedy algorithms, generally starting with the most used data item, then placing the next most used data item

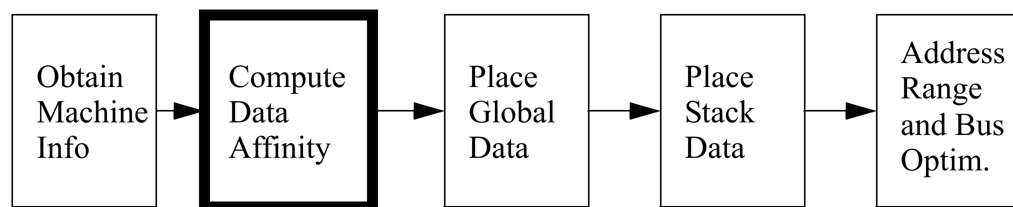
in relation to the first item, etc. Global optimization algorithms, such as grouping data items by mutual temporal affinity before final placement, have not been studied.

This research addresses these deficiencies.

Chapter 3

Data Affinity Analysis

Before discussing the topic of this chapter in depth, it will be helpful to have an overview diagram of the entire research. This diagram will be repeated in later chapters as a reminder of where each chapter's subject fits into the overview.



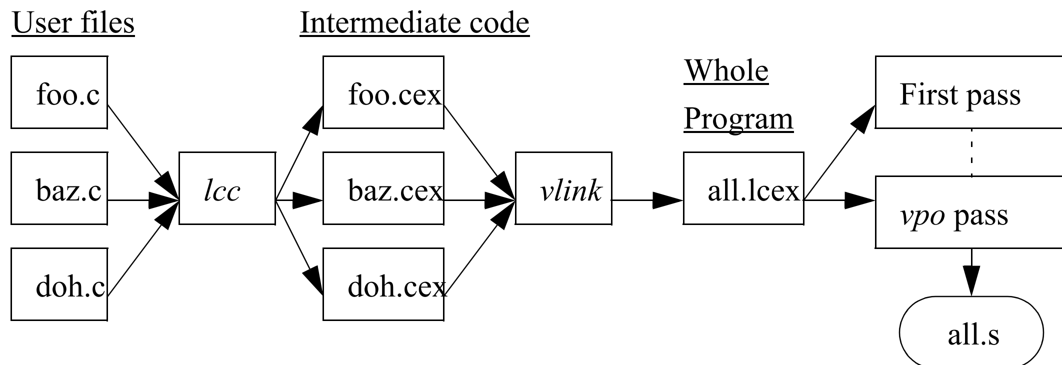
The first box in the diagram indicates that there are certain items of information about the cache hierarchy, such as size, line size, associativity, etc., that are needed for the other stages of the research. Discussion of how to obtain this information, along with discussion of the effort needed to retarget the analyses and optimizations from one machine to another, is deferred until Chapter 6.

Once this information is obtained, analyzing the user program that is being compiled can begin. This box is highlighted as it is the subject of the current chapter. The affinity information for global data that is computed in this stage is used in the next stage to place all global data. A much simplified data affinity computation is performed within the following box, in order to provide the information needed to place stack data (i.e. reorder local variables within each function in the program). Finally, the reordered local and global data enables the optimization of address computations and the minimization of bus cycles through memory access coalescing in the last box.

It should be noted that there are large time gaps in this diagram. The first box represents an offline process that takes place as part of the design of the compiler, not at com-

pilation time. A program is described in Chapter 6 to automatically provide most of the data needed.

When performing any whole-program optimizations, the compiler used in this research, *vpo* [Benitez 1994], is used in a multi-stage sequence, as shown in the diagram below:



The processing of the application program can be described at a high level, as follows:

1. The *lcc* compiler front end is used to check the correctness of the source code and translate each source code file into the corresponding intermediate code file.
2. A separate tool, called *vlink*, collects the intermediate code form of all files in the user program and merges them into a single intermediate code file. This tool is described later in this chapter.
3. A first pass over the intermediate code for the whole program builds a call graph for the program, builds a global variables symbol table, and annotates the intermediate code with lists of the loads and stores made in each basic block of the program. The latter operation is the first part of the data affinity analysis stage.
4. As the end of each function is reached in this first pass, a simplified representation of the function is produced, containing little besides the lists of loads and stores made by each basic block. A call graph node for the function is created, and the simplified representation of the function is attached to the call graph node.
5. After the end of the last function in the whole program is reached in this first pass, the call graph is processed for use by other optimizations besides the ones in this research. For example, arcs between call graph nodes indicate which nodes call other nodes. When this processing

is finished, the simplified representations of each function are inlined into their callers until eventually only the `main()` function remains. This function now contains a representation of the global data accesses for the whole program.

6. The whole-program simplified representation is now traversed in numerous paths, with each path used to record the temporal affinity of data items on the path.
7. Data placement takes place after data affinity analysis.
8. A second pass (the *vpo* pass) over the intermediate code for the whole program now takes place, after rewinding the whole program intermediate code file (all.lcex) to the beginning. This pass is the same as the only pass that normally takes place when no whole-program optimizations are performed. *vpo* applies all of its optimizations to each function in isolation during this pass. The dotted line connecting the first pass to the *vpo* pass indicates that data structures, such as the global symbols list, can be passed from the first pass to the *vpo* pass if they are useful in optimizations performed in the *vpo* pass.
9. At the end of each function in this second pass, *vpo* sets up the prologue and epilogue code in a machine-dependent way for the function, and then generates the assembly language form of the function into the output file (all.s). Just before the generation of assembly language, the placing of stack data, addressing range optimization, and bus cycle (coalescing) optimization are performed, as shown in the final two boxes of the diagram above.
10. The linker is called to translate the assembly language into an executable form (not shown).

In summary, the first box in the first diagram at the beginning of the chapter, *Obtain Machine Info*, is done offline. The second box, *Compute Data Affinity*, is done in pieces during the first pass over the whole-program intermediate code file. The third box, *Place Global Data*, is done at the end of the first pass over the intermediate code file. The final two boxes, *Place Stack Data* and *Address Range and Bus Optimizations*, are done at the end of the processing of each function in the second (*vpo*) pass over the intermediate code file for the whole program.

Now that the diagrams have been described and the analysis stage placed in the proper temporal context, the subject of this chapter, data affinity analysis, will be discussed in detail.

Introduction

In order to reduce conflict misses and compulsory misses in the memory hierarchy, the data placement algorithms will require a profile of the application program's data access pattern as input. This data profile must accurately reflect the *temporal affinity* of each data object for other data objects; that is, given that object A has just been accessed in the application program's execution, what other data objects will be accessed within a short period of time in the future? These other objects could evict object A from one or more levels of the memory hierarchy if their addresses cause a conflict. In order to place data so as to minimize cache and TLB conflicts, the temporal data reference pattern of the application program must be characterized. The goal is to produce an *affinity array*, in which an entry in row i and column j indicates that, after accessing the data object whose profile-assigned index is i, the application program accessed the object with index j a certain number of times within a given time and space window. Thus, the sum of the (i, j) and (j, i) entries in the array will depict the magnitude of the cost of assigning addresses to these two objects that conflict in the level of the cache hierarchy that is being profiled. Because different levels of cache have different capacities and possibly different policies on dealing with misses on data writes (e.g. the first-level data cache could have a write-through policy and not allocate a cache line when a write miss occurs, while the second-level cache could allocate a cache line on each write miss), an affinity array will be maintained for each level of cache. *Data affinity analysis*, hereafter referred to as *data analysis* for brevity, is the examination of the application program in such a way as to enable the building of the affinity array for each level of the cache hierarchy.

In this research, the focus will be on the placement of global variables. Placement of local variables will also be performed, but experience shows that few functions have enough local variables to exceed the capacity of the first-level data cache. Also, the original first-level cache research described by Calder et al. reported very little benefit from profiling stack (local) variables and trying to place the starting address of the stack to minimize conflict between global variables and stack variables [Calder et al. 1998].

Profiling of both code execution and data access patterns has been performed in various ways in prior research. These approaches will be summarized briefly in order to motivate the original approach taken in the present research.

3.1 Code and Data Profiling in Prior Work

Code profiling attempts to characterize the execution pattern of the application program at either the function level or the basic block level. Code profiling at the function level is exemplified by the tool *gprof* from the Free Software Foundation [Fenlason and Stallman 1992]. When requested by a command-line option, the *gcc* compiler instruments the executable code by inserting a call to a counter function at the beginning of each function. When the program runs, the counters keep a total of the number of times each function is executed and produce an output file of raw data counts. These counts are summarized by the *gprof* tool into a readable report. The raw counts file can be read by a compiler during a second compilation pass over the application program; for example, certain expensive optimizations could be targeted only to the most frequently executed functions.

At the basic block level, analysis is more difficult. Both *static* (i.e. during a single compilation) and *dynamic* analysis methods have been developed in past research. Some dynamic methods (i.e. methods that use instrumentation of the executable code, and hence require a *profiling run* of the application program to produce profiling data) simply extend the basic idea of *gprof* by inserting counters at the beginning of each basic block. At the end of the profiling run, the counters are written into a report file, which can then be related to the basic blocks in the original source code using debugging symbols in the executable file. The Quantify tool from Rational Software uses this method [Pure Software 1995]. Within our own *vpo* infrastructure, EASE (Environment for Architecture Simulation and Evaluation) provides efficient means to profile at the basic block level [Davidson and Whalley 1991].

It can be more useful for some compiler optimizations to know the sequences of basic blocks (i.e. the paths) that are executed most frequently, rather than a simple basic block execution count. The temporal relationship graph (TRG) that was discussed in Section 2.1 is a means of providing path information at the function level and is useful for code placement optimizations. At the basic-block level, path profiling has been studied in recent years [Ball and Larus 1996, Ammons and Larus 1998, Larus 1999]. The path profiling techniques have been refined to a practical level of efficiency, i.e. the profiling run takes 4-8 times as long as a normal program run. However, only a portion of the program is profiled,

and the results have not been applied to code or data placement. The TRG profiling run, as noted earlier, takes about 25 times as long as a normal program run, which reduces its practicality for most optimization clients. Recent work in Whole Program Streams [Chilimbi 2001] directly address data profiling for data placement and related optimizations by extending the previous path profiling methods to record streams of data references. Presumably profiling runs to generate Whole Program Streams are somewhat slower than the previous path profiling runs, but timings are not reported.

Our understanding of program behavior has been advanced by these dynamic profiling experiments, but they suffer a certain lack of practicality. Not only are the profiling runs quite lengthy for the methods that are useful in placement optimizations, but few software engineers have compiled, at any point in their careers, an application program with feedback from a prior profiling run used to guide optimizations. It is unclear that compiler optimizations that are dependent on any form of dynamic profiling feedback will ever have a widespread impact in the world of computing. As a result, the present research includes the development of novel *static* methods of profiling data access patterns.

Past research in static analysis has centered on the determination of basic block execution frequencies within a function. The compiler used in this research, *vpo*, has a static analysis subroutine that computes basic block execution frequencies as follows:

1. The entry block in the function is given a frequency value of 1.0; that is, it is executed once per call of the function. (All blocks have a value that correspond to the number of times they are expected to be executed on each call to their function.)
2. Any block with only one child block simply passes its frequency estimate to its child.
3. A block with more than one child evenly divides its estimated frequency value among its children. This means that it is assumed that the two branches in an `if-else` construct are equally likely to be taken, each `case` within a `switch` statement is equally likely to be executed, etc.
4. An estimated number of iterations for each loop is factored into each block within that loop as a final step. Nested loops thus have their blocks scaled up by a product of several iteration counts.

Research efforts in static analysis have focused on attaining greater precision than these "rules of thumb" provide. Making a prediction for a non-loop branch that is more pre-

cise than the 50-50 rule for an `if-else` statement has been achieved by developing heuristics based on instrumented observations of a large body of real application programs. Ball and Larus developed heuristics primarily based upon the two possible basic block successors of a branch [Ball and Larus 1993]. For example, if one block started a loop, while the other block avoided the loop, it was observed to be far more common to execute the loop rather than avoid it, so the 50-50 split between the two destinations of the branch statement should be altered to predict a far greater likelihood of entering the target loop. On the other hand, both calls and returns are more often avoided by branches than reached by branches, as they tend to happen often to handle errors and other exceptional conditions. The various heuristics were then combined into a single heuristic decision procedure. While not as accurate as the perfect branch prediction that could be provided by using the same input data for a dynamic profiling training run and the final measured run, their heuristic procedure was far superior to the 50-50 rule. This work was extended to predict the frequency of calls over the whole call graph of a program by Wagner et al. ([Wagner et al. 1994]), in addition to developing new intraprocedural branch prediction heuristics (e.g. if one arm of a conditional branch writes to a variable that is used elsewhere, that arm of the branch is more likely to be taken). Intraprocedural branch prediction was improved further by Patterson using data value range propagation to better determine which branches are likely to be taken [Patterson 1995]. Rather than simply trying to predict which branch is taken more than half the time, as in the other works discussed above, a probability is assigned to each direction of a branch.

In the present research, the existing *vpo* subroutine that uses rather simple rules for non-loop branches has been used. One enhancement was an analysis of the exact iteration count of each loop, where the iteration count is dependent on constants, as is often the case. This enhancement reduces the number of occasions on which the iteration count must be assigned the default value (presently set at 10).

3.2 Interprocedural Static Data Analysis

Where most prior research in static analysis has been intraprocedural (and the exception, by Wagner et al., did not attempt to profile data accesses), our static data analysis must be interprocedural. There are several reasons for this requirement:

1. Imagine a code sequence in which the following happens: global variable A is accessed, a function call is made, and then global variable B is accessed in the caller after the return from the called function. What is the sequence of global accesses? The answer depends on whether the called function accessed global variables. If it did not, then there is a direct potential for cache conflict between A and B. However, if the called function accessed other global variables and then called another function, which accessed yet more global variables and called another function, etc., then there is actually very little temporal affinity between A and B in this sequence. In fact, there might be so many global variables accessed between the access to A and the access to B in the calling function that there is little or no chance of A still residing in the cache when B is finally accessed, in which case B is not evicting A from any cache and has no conflict affinity with A.

2. If a function calls a second function from within a loop, then the intraprocedural static estimates of execution frequency for the basic blocks within the *callee* need to be scaled by the loop iteration count from the call site in the *caller*. As the callee might be called from numerous call sites scattered throughout the program, its basic block frequency estimates cannot simply be changed once and for all; they need to be scaled for each call site.

3. If a function appears to operate only upon its parameters and local variables, it might actually be accessing globals that were bound to its formal parameters at one or more call sites. Thus, we need to statically profile not just its global variable accesses, but its accesses to its own formal parameters, which then need to be translated into global variable accesses at any such call sites. This binding of actual to formal parameters obviously can be different at each call site, so there is no single profile of the global accesses of a given function.

Given these considerations, it is apparent that we need to profile the following within each function:

- Accesses to statically allocated global variables.
- Accesses to formal parameters.

- Calls to other functions.
- The passing of a list of actual parameters to a callee.
- Recording the loop nest structure within the function, including which basic blocks are in which loops, which loops are within other loops, and the iteration counts of each loop.

The restriction to statically allocated global variables (hence the exclusion of heap-allocated global variables) is made due to the high engineering cost of profiling heap accesses. The prior work in cache-conscious data placement concluded that this portion of their infrastructure was the greatest amount of work for the least amount of benefit on many benchmarks, although it was critical for heap-intensive benchmarks [Calder et al. 1998]. Rather than duplicate their work, it was decided to focus on original ideas in data analysis, data placement, and related optimizations. Benchmarks that are heap-intensive will be excluded from the present research. Future extensions could include analysis and placement of global heap-allocated variables.

Having profiled these items within each function, and given the intraprocedural basic block execution frequency estimates, the raw data will be available to perform inter-procedural data analysis. We now examine in more detail the analysis within each function, followed by the interprocedural analysis. Before doing so, a high-level pseudocode will be shown for the entire data affinity analysis infrastructure, to serve as an overview of the remainder of the chapter:

1 Calculate execution frequency estimates for each basic block in the current function.

2 Create a list of loops in the current function, with each loop containing a list of the basic blocks it contains, and a formal parameter list for the function.

3 Annotate each basic block in the function with a list of global variables and formal parameters accessed, function calls made, and an actual parameter list for each function call. Record for each access to a global variable of formal parameter whether it was a load or a store.

4 At the end of the processing of the function, transfer the information from step 3 above into a data structure (called a Control Flow Graph, or CFG, hereafter) much smaller than the basic block data structure, which will include the execution frequency estimate for the block, the access and call information for each block, and the name of the function within which the basic blocks were found.

5 Attach the data structures created in steps 1, 2, and 4 to the newly created call graph node for that function.

6 After all functions have been through steps 1-5 above, traverse the CFGs within each call graph node and record global variable indices for global variables accessed, and pointers to the call graph nodes for functions called. Eliminate calls to functions that were marked as library functions. Simplify and prune the CFG for each function.

7 Inline all the CFGs for all functions into their callers, replacing formal parameter accesses within the body of the inlined function with the corresponding actual parameters as the CFG is inlined. Free the callee's CFG from memory after it has been inlined into all callers.

8 After all CFGs have been inlined and only the main() function remains, perform path analysis to detect temporal affinity among global variables. Record this temporal affinity in the affinity arrays for the different levels of cache in the target system.

The first step in the pseudocode is performed using the *vpo* function described in Section 3.1 above. Steps 2-3 are discussed in detail in section Section 3.2.1 below. Steps 4-6 are discussed in Section 3.2.2 below. Step 7, inlining the CFGs, is described in Section 3.3. Finally, step 8 is discussed at length in Section 3.4.

3.2.1 Intraprocedural Analysis

Analyzing accesses to global variables within each function is fairly straightforward. The RTL (register transfer list) encoding of the program that is maintained by *vpo* [Davidson and Fraser 1980] distinguishes between globals and locals. There are a few hurdles to overcome. First, each access to a global must be analyzed for its level of indirection. At level zero, the address of the global is being taken in the RTL, but the memory location for the global is not accessed. This should produce no entries in the global data profile. At level one, the value of the global is loaded or stored. This is the primary type of access that is recorded in the data profile. At levels two and higher, the value of the global is used as a pointer to another location in memory and dereferenced (and this could then be dereferenced again recursively). This access is also recorded in the data profile, but unfortunately *vpo* does not possess the sophisticated pointer analyses that would be required to also record the pointed-to memory location in the data profile.

The data structure in which the accesses to global data are recorded is a simple linked list attached to each basic block. Each entry in the list records the name of the global accessed, the index of that global within a global symbol table, and whether the access was a load or a store.

Analyzing function calls and formal parameters is similar. A function call is given the symbol table index of -1 to flag it for later processing, while a formal parameter is given an index of -2. A key difference between accesses to globals and to formal parameters is that the latter *must* be accessed at a level of indirection of two or higher in order to be of interest in global data analysis. This is because an actual parameter that is passed by value is only accessed once, at the time it is placed in an argument register or stack location for the function call. After that access, the references to the formal parameter within the callee function do not cause memory accesses to the location of the original global variable. However, a global variable that is passed by reference causes the formal parameter dereferencing operations in the callee to be accesses to the global variable in memory. Thus, the latter references are of interest to later interprocedural data analysis, while direct accesses to the value of a formal parameter are not of interest and are not recorded.

Analysis of the setup of actual parameter lists presents some difficulties. After the optimizer in *vpo* performs register allocation, a formal parameter might be held in a register throughout the body of a function and its name would therefore never be visible. This cannot happen for global variables, as it is not safe to hold a global in a register for any region of code that has stores to memory unless sophisticated points-to analyses are available to ensure that the value in the register has not become stale due to an indirect store to the location of the global variable that is held in the register. Because *vpo* does not yet have such analyses, globals are held in registers for relatively short durations, and never across the entire body of a function. The name of the global will always appear at least once in the RTLs of any function that accesses it, while a formal parameter name might disappear entirely.

Due to these complications, the primary work of global data analysis is performed early in the optimization sequence in *vpo*, just before register allocation is performed for the first time. With all parameter and global names visible, it is easier to analyze the data access patterns at this point. Examination of the RTLs throughout the optimization

sequence indicates that, while the addressing arithmetic associated with accessing a global might be moved out of a loop in a later optimization phase, the accesses within the loop will remain in the form of indirect addressing through a register. Thus, the later optimizations rarely get the opportunity to eliminate global accesses, although they eliminate most of the accesses to locals, which do not concern us at this point in our analysis.

Building a lengthy list of actual parameters, and associating the list with their function call, has proven to be difficult. The problem was finally solved by processing actual parameters in three stages. First, the *lcc* compiler front end, which is used by *vpo* for most of its target machines, was modified to record the information about actual parameters that is visible in the front end of a compiler before any optimizations or transformations to a lower-level representation such as RTLs. The information is recorded in a *metadata* line, which is a comment line in which the front end can pass information to the back end of the compiler (*vpo* being the back end in this case). For globals and formal parameters, it is usually straightforward for the front end to determine the level of indirection of the access that passes them as actual parameters. If the level of indirection is such that the actual parameter interests us (i.e. it is a by-reference parameter), then the internal numeric code (for a global variable) or the ordinal number (for a formal parameter, e.g. the first formal parameter has ordinal number zero, etc.) is recorded in the metadata line. If the actual parameter is a local or constant or has a level of indirection that makes it not of interest to global data analysis, the exclamation point character is recorded as a pseudocode to distinguish it. If the expression that builds the actual parameter is too complicated to analyze in the front end, it is also flagged with the '!' character. These actual parameters are referred to as "don't-care" parameters henceforth.

It sometimes happens that code simplifications in *vpo* will make an actual parameter analyzable during our intraprocedural analysis. In that case, the '!' is replaced by the global code or formal parameter ordinal number. Thus, a second chance at completing the list of actual parameters has been obtained. Still there might be difficulties in analyzing the building of an actual parameter. For example, it has been observed that if the address of a global is taken and then passed in two or more actual parameter positions, as in the function call: `foo(x, y, &bar, z, &bar);` then the front end will hold the address of the global in a temporary local variable and pass the value of that local variable twice. This will

deceive our analyses into thinking that there is nothing of interest in these parameter positions. In order to resolve this problem, a pass through the RTLs is made after register allocation has been performed. The local temporary will now be allocated to a register in the above example, and the contents of that register can be traced backward through the RTLs until we find the address of "`bar`" being placed in it. Thus, on the third chance at analyzing these actual parameters, success is achieved. With these three efforts at building a list of actual parameters, our observations have indicated no cases of failures to profile items of interest in actual parameter lists.

After a list of actual parameters has been built, the next function call RTL causes a function call data profile entry to be made in the profile list for the current basic block. That function call data profile entry has the list of actual parameters attached to it at this point. The "third chance" analysis will take this list and attempt to complete it after register allocation.

All that remains is to build a data structure that reflects all of the loops, nesting included, and which basic blocks belong to each loop. This data structure is created in a compatible form by the loop analyses in *vpo*. Because those loop analyses are not performed at this early stage of optimization (prior to register allocation), a customized version of the loop analyses was created as part of this research. The needed information is then transferred into a more compact data structure that contains only the information needed in our analysis.

3.2.2 Interprocedural Data Analysis

The first step needed to perform interprocedural data analysis is to have a data structure that spans all functions in the program, so that the intraprocedural data profiles can be associated with their functions prior to interprocedural analysis. The prior research efforts of Jason Hiser have contributed a program-wide call graph data structure, with items for each function and arcs connecting callers to callees [Zhao et al. 2002]. The present research uses this callgraph as a means to associate intraprocedural data profiles with the functions from which they were obtained. The connections between functions are represented by function call data profile entries, so the existing call graph arcs will simply be ignored by this research, although they are useful for other optimizations. A prerequisite to whole-

program analysis via the call graph is a representation of the entire program in a single intermediate code (i.e. RTLs) file, with variable and code label renaming to avoid conflicts between names that were formerly in separate files. Hiser also designed a tool (called *vlink*) to merge intermediate code output files from the front end into a single file with renaming of labels and global variables. As part of the current research, this tool was enhanced to rename formal parameters to keep their names distinct across the whole program.

During a first pass over this single, merged intermediate code representation of the whole program, the call graph is built as each function is processed. In order to have an accurate representation of the program as it will be when *vpo* has finished optimizing it, the normal sequence of *vpo* optimizations is applied. It is during this pass that intraprocedural data analysis is performed, first before register allocation, then later after register allocation in order to make the actual parameter lists as complete as possible. When the end of a function is reached, a call graph node is created for that function. At this point, our code attaches the global data analysis data structures to the call graph node. These data structures consist of:

- A formal parameter list, built easily from the declaration lines, one per formal parameter, that were emitted by the front end at the beginning of the function.
- A loop list data structure, built as described previously during intraprocedural analysis.
- A CFG (control flow graph) data structure, to be described below.

The intraprocedural data profile entries were attached to each basic block in the internal *vpo* representation of the function. This internal representation is about to be freed in memory as *vpo* prepares to process the next function. It consumes a good deal of storage per basic block, as there is much to record of interest to *vpo* in various optimizations. As a result, a very reduced form of the basic block graph is created, which will be referred to henceforth as the CFG (control flow graph) for data analysis. Each CFG node has children links to match the basic block graph from *vpo*, a copy of the data profile entries for the basic block, and the static intraprocedural execution frequency estimate field from the basic block. One simplification is to remove all back edges while creating the CFG. Back edges are useful for loop analysis in the basic block representation of a program. For our purposes, all useful

looping information has been captured in the loop list and in the static execution frequency estimate fields, which were scaled up by the enclosing loop iteration counts. To avoid infinite looping in our analyses, back edges will never be followed and can therefore be eliminated.

With all essential information transcribed from the basic blocks into the CFG, *vpo* can now free all memory associated with the function and process the next function.When the last function in the file has been processed, the callgraph processing needed for other interprocedural optimizations is performed, such as building the arcs from one call graph node to another and weighting these arcs with the estimated call frequencies. A file of standard runtime library function names is read, and call graph nodes for library functions are flagged. Functions that call themselves recursively, whether directly or indirectly through a chain of other functions, are marked accordingly. After this processing is completed, processing for global data placement resumes.

Simplifying and Pruning the CFG

Now that call graph processing has completed, certain simplifications to the CFG data structure can be made. Calls to runtime library functions are removed, as these functions rarely use global data and are not analyzable for our purposes. Data analysis entries for function calls can be linked directly to the call graph node for the callee to facilitate interprocedural analysis. In order to reduce memory requirements, a first pass at pruning the CFG can be made.

The two steps to pruning the CFG for a function are bypassing irrelevant nodes, and removing unreachable nodes. When a child node has an empty data profile list, indicating that the corresponding basic block made no accesses to global variables or formal parameters and made no function calls, then it might be possible for the parent node to bypass that node. There are several cases that must be considered, because each CFG node can have either one or two children. In *vpo*, the left child of a basic block is the unconditional successor, reached either by an unconditional jump or a simple fall-through to the successor. The right child is the conditional successor, reached only if a conditional branch is taken. Every basic block except the bottom block of the function has a left (unconditional) child, but not every block has a right (conditional) child.

The first case of an irrelevant child node occurs when the child node is the left child of a parent that has no right child, and the child node has an empty profile list. The parent node can simply make the left child of the irrelevant node its new left child, and the right child (if any) of the irrelevant node becomes its new right child.

The second case of an irrelevant child node occurs when the parent node has both left and right children, but the child with the empty data profile list has only a left child. The parent node can then bypass the irrelevant node and point to the only child of that irrelevant node instead.

These two cases of bypassing an irrelevant node can be seen in Figure 3.1 below. In both drawings, parent node A bypasses node E, which has an empty data profile list, and ends up with nodes B and C as its children. The dotted lines indicate connections that are still present but unreachable, as their parent node has been bypassed. Figure 3-1(a) corresponds to the first case above, while Figure 3-1(b) matches the description of the second case.

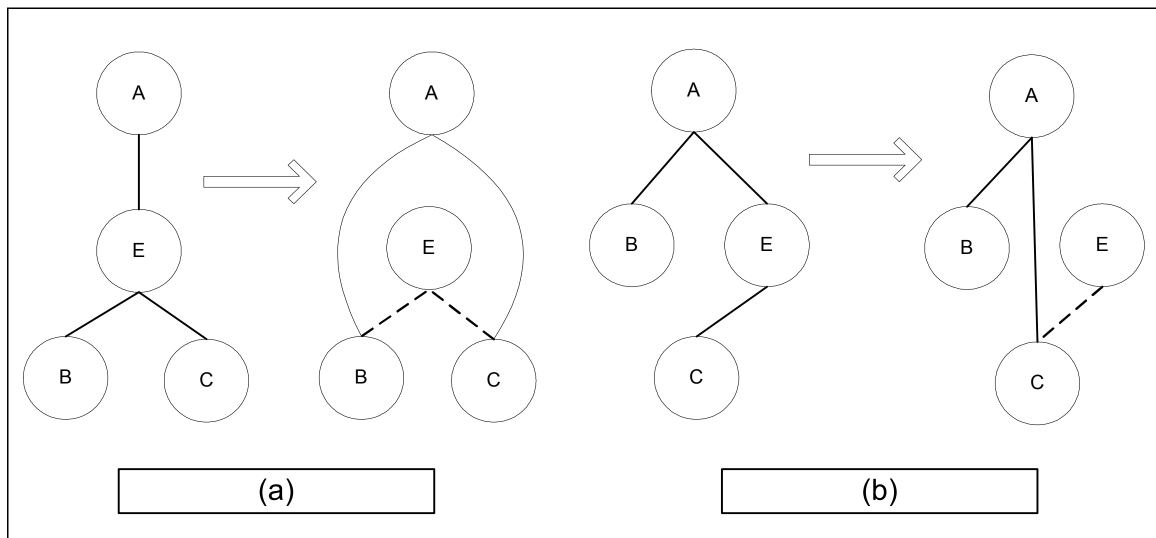


Figure 3-1.Pruning CFGs.

In order to simplify the CFG further, some additional bypassing of nodes is also performed. In Figure 3-2(a), we see a case that does not fit either case previously described, because parent node A has two children, each of which has two children. However, one of the two children nodes has the other as its child. In this figure, node E has an empty data profile list. As a result, no path from A to E to B contributes anything to our path analysis

that will not already be recorded when we analyze the path from A directly to B. Therefore, if A is the only parent of E, the connection from E to B can be removed. Removing the connection reduces this case to the case in Figure 3-1(b), and A can now bypass E.

One possible result from bypassing children nodes is shown in Figure 3-2(b). Both children of parent node A had empty data profile lists. After bypassing E1 and E2, node B will become the left child and the right child of A. In this case, the right edge from A to B will be removed to simplify our later path analysis, and the subgraph will become just node A with node B as its left child.

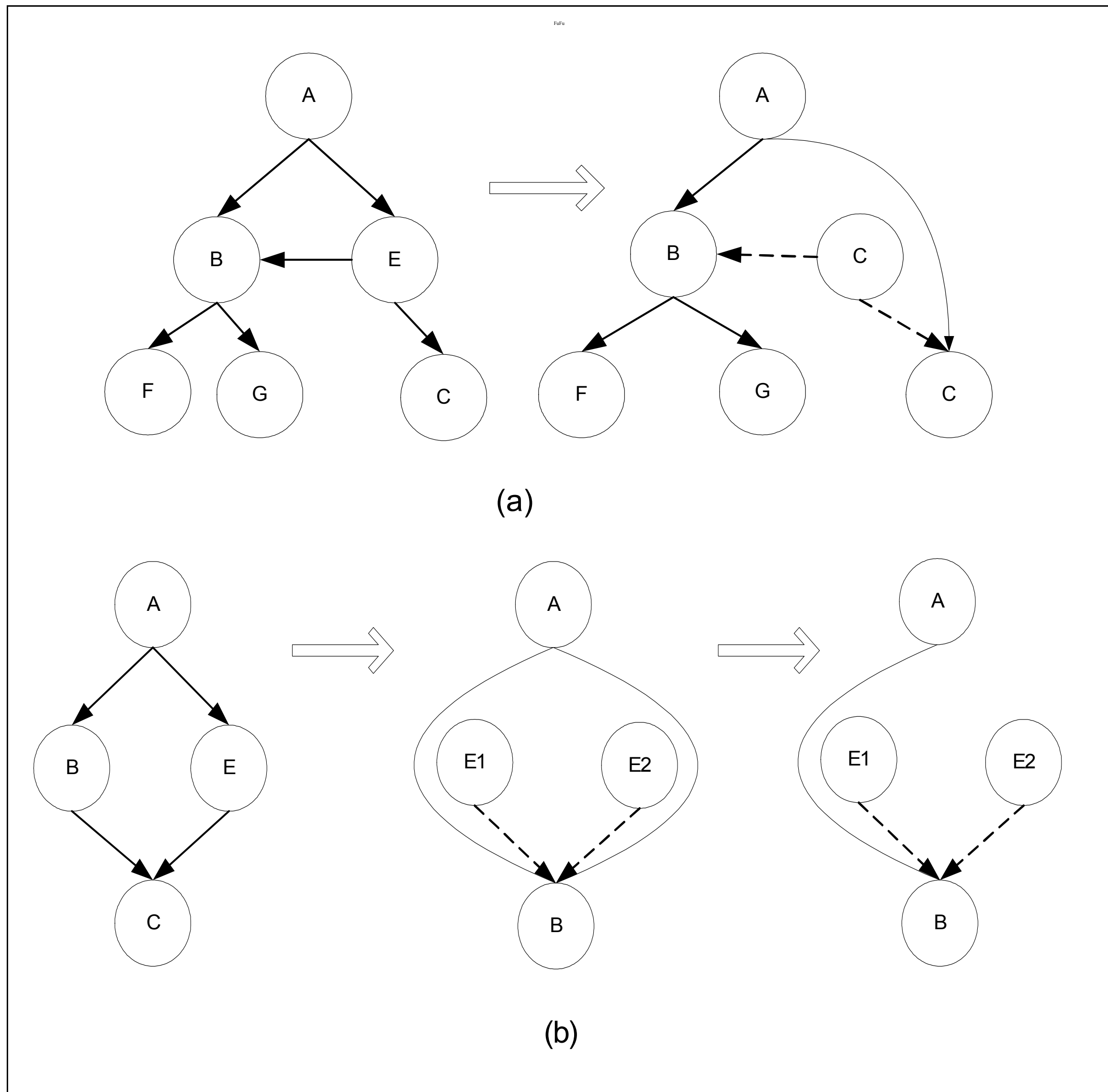


Figure 3-2.Further CFG pruning examples.

There is one guard condition for all bypassing cases. Because we are preparing for a later analysis of paths, described in the section below, care must be taken to preserve the loop structure of the program. In particular, data items accessed together within a loop have greater potential to *thrash* (conflict on every access) in a cache than data items accessed in separate loops. For example, in the pseudo-code fragment below:

```
for (i = 1 to 200)
   access global array element A[i]
   access global array element B[i]
endfor
```

If global arrays A and B were mapped to the same starting data cache line address (i.e. the same address modulo the cache size), then each access in this inner loop would cause a data cache miss. On the other hand, in the following pseudo-code fragment:

```
for (i = 1 to 200)
   access global array element A[i]
endfor
for (j = 1 to 200)
   access global array element B[i]
endfor
```

If global arrays A and B are mapped to the same starting data cache line address, each will at least benefit from spatial locality, i.e. even if accessing B[0] causes a data cache miss because A has evicted B from the cache, the cache line fetch will bring in several elements of B, so that the next several iterations of the second loop will cause hits in the data cache. If outer loops enclose these inner loops in both examples, then A and B definitely have temporal affinity and their potential cache conflict must be noted in our affinity array. However, it must be recorded that the first example is a more severe cache conflict affinity than the second example.

In order to keep this difference in severity from disappearing, we must avoid any accidental merging of loops via node bypassing. In the second example above, a CFG node will exist for the first loop and another for the second loop. A third CFG node will fall in between the two loops; it serves as the fall-through node when the first loop exits (i.e. this was the structure of the original basic block graph in *vpo* from which these CFG nodes were

derived). If there are no enclosing outer loops for the loops that are shown, then this middle node will have an execution frequency estimate value of 1.0 or less, as it has no loop iterations to scale it up to a higher number. It will often be the case that no accessing of globals occurs between the loops, as in the pseudocode examples, which would ordinarily make such a node a candidate to be bypassed. However, bypassing the node that separates the two loops will make the two loop CFG nodes consecutive, and later path analyses will move from the first loop to the second loop with no warning that there is no conflict potential in the second code fragment, unlike the first code fragment with a single loop. (A more detailed discussion of loop-based conflict analysis appears in the section below.) We avoid obscuring the separateness of two loops by refusing to connect a parent node with a frequency estimate value of more than 1.0 to a grandchild node with frequency estimate value of more than 1.0 (i.e. two CFG nodes that must be inside loops) if the child node between them that would be bypassed has a frequency estimate value of 1.0 or less.

After a bypassing sweep through the CFG nodes for a function has been performed, a flag is reset to FALSE in each CFG node. Because the CFG nodes are linked with a "next" pointer in addition to the right and left child pointers, all CFG nodes are reachable, even if they have been bypassed by their parents. Each function has a top CFG node (which has no parents) and a bottom CFG node (which has no children). Then starting from the top node, the left and right child pointers are followed, setting the flag to TRUE for each CFG node reached. Upon completion of this process, the CFG nodes are examined by following the "next" pointers, and any CFG node with its flag set to FALSE is unreachable from the top node and is pruned from the CFG and its memory recovered. The "next" pointers are made to bypass the pruned node. Note that the special top and bottom nodes are required to maintain the structure of a CFG and cannot be bypasses or pruned, regardless of the emptiness of their data profile lists.

3.3 Inlining Functions in the CFGs of a Program

As noted earlier, a function could make reference only to its formal parameters, yet have an effect at runtime on the global variable access patterns of the program. In order to make the connection between formal and actual parameters, a list of formal parameters has been

assembled for each function, and a list of actual parameters attached to each function call entry in the data profile lists for their CFG nodes. By *inlining* a function's CFG into the call sites of its callers, the correspondence between actual and formal parameters can be made explicit by substituting actual parameter names for their corresponding formal parameters throughout the cloned CFG of the callee function as it is inlined. A second benefit of inlining CFGs is just as important. As the inlining is performed, the execution frequency estimate values of each inlined CFG node are multiplied by the frequency estimate of the calling CFG node. This scaling reveals the true impact on the memory hierarchy of each CFG node as inlining continues until the entire program representation has become a single function's CFG (i.e. for the `main()` function in the C language, or the name of the program entry point in other languages). At that point, there are no more formal or actual parameters in the program representation, and path analysis can commence, as described in the next section.

Inlining CFGs is a memory-intensive activity, so there exists the possibility that virtual memory could be exhausted before the inlining of all functions into the CFG for the `main()` function. Several aspects of the design of the inlining code take memory efficiency into account. Inlining begins by making a single copy of each directly recursive function into each of its recursive call sites. This is done first to make copies of the function before it has grown in size by having its non-recursive callees inlined into it. The purpose of inlining directly recursive functions a single time is to expose the temporal affinity between the global variables that are accessed at the end of one execution of the function and the global variables accessed at the beginning of the next execution of the function. It would be more precise to inline more than once recursively, if we knew the approximate depth of the recursive call chain. However, that depth is not easily analyzable, the initial exposure of temporal affinity by a single inlining is more significant than repetitive exposures of that affinity, and inlining more than once in succession would cause an explosion in memory consumption. After a single copy of the function is made at each of its recursive call sites, the newly produced recursive calls are deleted, and the function is no longer directly recursive.

Each function is scanned, and the number of call sites in each function is recorded in the call graph node for that function. Leaf nodes have zero call sites and are inlined into all their callers next. After they have been inlined, there is no further need for their CFG

representation, which is deallocated from memory. A new set of functions might become leaf functions after the original leaf functions were inlined; they are inlined next and deallocated from memory. Whenever inlining causes a function to become directly recursive for the first time, it is immediately inlined once, before it can grow any larger.

After each inlining performed, the bypass and prune sweeps are performed over the caller's CFG. This can remove nodes such as the callee's top and bottom nodes, which are not the top and bottom nodes in the caller and hence do not need to be preserved if their data profile lists are empty. More numerous are CFG nodes which only accessed formal parameters, and those formal parameters were matched to an actual parameter that is irrelevant to global data analysis, such as a constant or local or a pass-by-value actual parameter. These nodes now have empty data profile lists. In some cases, bypassing of empty-list nodes was inhibited in an earlier bypass-and-prune stage because the CFG node with the non-empty data profile list prevented the subgraph around it from falling into any of the cases described earlier. Now that the don't-care variety of formal parameters (i.e. those that were given a pseudo-name of "!" to mark them, as described in Section 3.2.1) have been deleted, a cascade of bypassing opportunities has been occasionally observed on real programs. The process continues until there are no more leaf functions to inline, and all callers have been minimized by pruning.

If all functions have not been inlined into the `main()` function, then the remaining functions, none of which are leaf functions, compose one or more indirectly recursive chains of functions. Starting with the function with the fewest remaining call sites (i.e. closest to being a leaf node in the call graph), functions are inlined into their callers, and any resulting direct recursion is dealt with immediately by a single inlining. This process will resolve the indirectly recursive function chains back into `main()`.

While the CFG data structures have been designed to minimize memory consumption, and freeing of memory is performed at the earliest possible time at each step of the process, there is still the possibility that, for very large applications with millions of lines of code, the memory of the host system for the compilation will be consumed, virtual memory paging from the hard disks will commence, and eventually the compilation will take an excessive amount of time or even fail due to exhaustion of the virtual memory. In order to prevent this problem, a space budget and a time budget are used to limit the inlining pro-

cess. As inlining proceeds, the total number of CFG nodes increases due to the typical function having more than one caller on average. A limit on total nodes is currently set at 600,000. A time limit on the inlining process is set at 480 seconds (8 minutes). These values were chosen by empirical observations on our typical host system for the compilation, a 500MHz UltraSparc IIi processor with 1GB of physical memory. The values can easily be passed to *vpo* as command line switches for other host systems if different values are needed. The benchmarks that will be discussed in the Measurement and Evaluation chapter are all able to be inlined completely into `main()` well within these limits, although some of the larger SPECint2000 benchmarks ([Standard Performance Evaluation Corporation 2002]) are cut short. The implications for path analysis of terminating the inlining process will be discussed below.

3.4 Path Analysis to Produce the Affinity Array

Dynamic methods of profiling paths through programs focus on the paths that are most commonly taken, based on code instrumentation and profiling runs. The usual approach is to construct a list of the "hottest paths" until the list covers some threshold percentage of the program runtime, such as 80% or 90%. The program-wide CFG, which has been constructed by inlining all functions' CFGs until the `main()` function contains a representation of the whole program, includes all program basic blocks that access global variables, whether or not those basic blocks are part of a hot path. However, there is a static frequency execution estimate for each CFG node, permitting the construction of hot paths from static analysis estimates. Terminating path analysis when a certain percentage of the program execution time has been covered is not a concern for static path analysis. This is primarily due to the speed of static path analysis, and partly due to the fact that the temporal affinity recorded by static path analysis is proportional to the execution frequency estimates. Thus, following a less important path records less affinity than for a hotter path. The global data placement should not be adversely affected by having more temporal affinity information, as long as the affinity is kept proportional to the true runtime significance of that affinity.

The primary concerns in choosing to terminate a particular static analysis path are loop structure and cache capacity. It was mentioned when discussing the code fragments in Section 3.2.2 that if global arrays A and B are accessed in two separate loops, and no outer loop encloses those two loops, then these two loops do not cause the dynamic lifetimes of A and B in any data cache to overlap. That is, when the first loop is finished processing global array A, there will be no thrashing between A and B if the second loop loads elements of B that evict A from a level of cache. (There might be other program regions that cause A and B to have cache conflict affinity, but not these two loops.) Therefore, the bypassing and pruning stage avoided pruning CFG nodes that separated outer loops. Similarly, a static path profile must not continue from one outer loop to the next, passing through a CFG node that is not contained in any loop and which separates those two outer loops.

The cache capacity concern can be understood intuitively as follows. Suppose that a lengthy outer loop encloses multiple inner loops that access a large number of global data items. Early in the outer loop, global variable A, with a size of 4KB, is accessed. Much later, global variable B, with a size of 2KB, is accessed. The level of data cache for which which we are analyzing temporal affinity has a capacity of 16KB. After accessing A, a particular path follows various branches until it reaches the much later accesses to B. If the path between A and B accesses exactly 16KB of other data items besides A and B, then it is possible that these data items have evicted A from the cache. It is also possible that if only 4KB of data had been accessed in between A and B, that the final data placement would have placed these items in direct conflict with A (perhaps because other paths were much more significant than this one), and A would be evicted before reaching B. The likelihood of A being evicted is not known at analysis time, because the final placement has not been determined, but it seems to be intuitively proportional to the amount of unique data accessed between the accesses to A and the accesses to B. If 200KB have been accessed since A was last accessed, it seems almost certain that A no longer resides in the cache. In any case in which A no longer resides in the cache, it is misleading to record any temporal affinity between A and B, as B cannot evict A from the cache if A has already been evicted from the cache by other data accesses. Where in the interval between the size of A and some large multiple of the size of the data cache should temporal affinity between A and another variable no longer be recorded? Past work settled on the figure of twice the cache size as

the best point to ignore the possibility of further cache conflicts [Hashemi et al. 1997]. This number was obtained through empirical trial and error. This research will conduct its own trial and error measurements in this regard, but will start with the same cutoff point of double the cache size.

With these considerations in mind, the design of a method of static path analysis can be described. CFG nodes that separate different outer loops, and which therefore are not contained in a significant outer loop themselves, are signified by a static execution frequency estimate value of 1.0 or less. (It is true that branching could reduce the frequency estimate values by a factor of 0.5 in succession until a tiny value is reached, at which point even CFG nodes contained in loops could still have values of 1.0 or less. However, such regions cannot possibly be important when viewed in the context of real application programs which contain CFG nodes that have execution frequency estimate values of more than 1 million.) Therefore, any analysis path that reaches a node with a small frequency estimate (currently set at 1.2 to provide a small margin above 1.0 and ensure that we not follow irrelevant paths) simply terminates. Likewise, when the path has seen a total data size of two times the capacity of a cache, affinity is no longer recorded for that cache on this path, although analysis can continue down this path if there are larger caches whose capacity has not been reached. Note that data size does not count duplicate references along a path; thus if we access global arrays A, B, and C according to the reference sequence ABCBCBCBC.... the total size computed is just the sum of the three arrays.

For each CFG node in the inlined CFG for the `main()` function that has a frequency execution estimate of 1.2 or greater and is contained within at least one loop and has a non-empty data profile list (i.e. it accesses global variables), a new static analysis path is begun. For each global variable accessed in this head node for the path, affinity is recorded in element (i, j) of the affinity arrays for each cache level if global i is accessed before global j in the data profile list. The affinity arrays are thus asymmetric and reflect the access order of globals. This ordering information will be used later to pack scalars into cache lines in such a way as to increase the likelihood that a cache miss is requesting a data item that is earlier in the cache line rather than later, in case the hardware is able to service earlier addresses more quickly. If a level of cache has a no-write-allocate policy, then elements in the data profile list that are tagged as stores will not be profiled in the temporal affinity array for that

level of cache. Note that static analysis needs to know two machine-dependent parameters for each level of cache: the size and write-miss policy. The determination of these values for different machines is discussed in depth in Chapter 6.

How much affinity should be recorded in the (*i, j*) element of each affinity array? To answer this question, we need to contemplate the relationships among loop structure, cache line and subblock sizes, and cache misses. Examine the code fragment below:

```
for i = 1 to 100
   for j = 1 to 160
      access A[j]
   endfor
   for j = 1 to 48
      access B[j]
   endfor
endfor
```

The parameters of an UltraSparc-IIi SunBlade 100 will be used in this discussion: 32 byte first-level cache lines with 16-byte subblocks, 64 byte second-level cache lines. It is assumed that the elements of arrays A and B are 4 bytes each, and that array A has 160 elements and array B has 48 elements, each of which is accessed after one trip through the two inner loops that access A and B (this is chosen because it is a common access pattern). Array A occupies 40 subblocks and 20 blocks (lines) in the first-level cache, and 10 lines in the second-level cache. Array B occupies 12 subblocks and 6 blocks in the first level cache, and 3 lines in the second-level cache; it is exactly 30% of the size of A.

In the analysis path that passes through the CFG nodes representing this loop nest, A will precede B in access order. Thus, only one entry in each affinity array, corresponding to (A, B), and referred to henceforth by the indices of A and B into the array as (*i, j*), will be increased by this particular path profile. (It is possible that B will occur before A on some other path and cause the (*j, i*) entries in the affinity arrays to be incremented.) The execution frequency estimates for the CFG nodes that contain the accesses to A and B will be 16000 and 4800, respectively, if we assume that there is no outer loop other than the one shown. A naive approach would be to increment the (*i, j*) affinity array entries by 4800 on the assumption that, during the program execution, there will be 4800 accesses to elements of B, each of which could evict an element of A if arrays A and B conflict in the final data placement. This would ignore the spatial locality of the caches. On each iteration of the

outer loop, it could be that some elements of A have been evicted from the cache and replaced by elements of B. However, each first-level cache miss in A will cause an entire subblock to be loaded into the first-level cache, which means that four elements of A are brought into the first-level cache. It is not possible to miss on every element of A in the loop nest shown. (Note that path analysis stops for a given level of cache when it encounters loop accesses of a global that is large enough to wrap around the cache, so capacity-induced self-evictions are not an issue in the path analysis design.)

To miss on every access, arrays A and B would have to be accessed in the same innermost loop in the loop nest. Then it would be possible for a miss in A to bring in an entire subblock, which would be evicted immediately by an access to B that maps to the same subblock (in the worst case, which is what we are trying to profile). Therefore, if A and B share all loops in the loop nest, then the (*i, j*) entry in the affinity arrays really would be incremented by 4800, but in the loop nest shown, this affinity must be scaled down by the number of 32-bit data items that fit in the subblock size of the first-level cache. So, 1200 is added to the (*i, j*) entry of the affinity array for the first-level cache. Note that with 64-byte second-level cache lines, there is a single second-level cache miss possible in this loop nest for every four misses in the first-level cache, as four 16-byte subblocks fit into one 64-byte second-level cache line. Only 300 second-level cache evictions of A by B can occur, so 300 is added to the (*i, j*) affinity array entry for the second-level cache.

Next, examine the code with the order of the inner loops reversed:

```
for i = 1 to 100
   for j = 1 to 48
      access B[j]
   endfor
   for j = 1 to 160
      access A[j]
   endfor
endfor
```

The danger in interpreting this code sequence with respect to cache evictions is that the larger array comes second, which causes the execution frequency estimates to be 4800 for the first CFG node with a global access, and 16000 for the second. However, as A is larger than B, it cannot evict a number of cache lines holding B that equals its own size. For example, A occupies 10 lines in the second-level cache, while B only occupies 3 lines. If

we applied the proportioning described for the first code fragment, we would increment the second-level affinity array by 1000 in the (*j, i*) entry, even though A is larger than B and must be evicting variables other than B on at least 70% of its accesses; A cannot cause more than 300 second-level cache line evictions of B. This is handled by scaling down the value added to the affinity array entry whenever the global variable that is accessed earlier is smaller than the global variable accessed later. The scaling factor is the ratio of the size of the smaller variable to the size of the larger variable, which is 0.3 in this case. Thus, instead of adding 1000 to the second-level cache affinity array for the (*j, i*) entry, only 300 is added.

Finally, consider the modified form of the original loop nest shown below:

```
for i = 1 to 100
   for j = 1 to 160
      access A[j]
   endfor
   for k = 1 to 20
      for j = 1 to 48
         access B[j]
      endfor
   endfor
endfor
```

Now, a loop has been added that encloses the loop that accesses B. For whatever reason, it is necessary to make twenty passes through B in order to perform the processing needed at this stage of the program. This causes the execution frequency estimate of the CFG node that accesses B to be multiplied by 20. However, B cannot evict any more elements of A from any cache than it did in the original code. To compensate for this problem, the loop lists for the CFG nodes that access A and B are compared. The iteration count for non-innermost loops that enclose the access to B but not the access to A is divided out of the affinity value recorded in the (*i, j*) entries in the affinity arrays. In the above case, this means that a factor of 20 is divided out, leaving the execution frequency estimate for the CFG node that accesses B at its original value of 4800 rather than the inflated value of 96000. The other scalings proceed as previously described, and 1200 is added to the (*i, j*) entry in the first-level cache affinity array, and 300 added to the corresponding entry in the second-level cache affinity array.

After recording the mutual affinity of globals within the head CFG node for a new path, the static path analysis function begins to recurse through the CFG. Recall that each CFG node has a left (unconditional successor) child and an optional right (conditional successor) child, and that back edges for loops have been removed. For each child that exists, a recursive call is made, passing in the list of globals from the head node, the current accumulated size, the capacity constraints for each level of cache (currently double the actual capacity of each cache, as discussed above), and the loop information data structure for the head node. For each global seen in the path, affinity is recorded from each global in the head node to each global in the path. If an access in the path is to a global that was in the head CFG node for the path, then the global within the head CFG node is marked as *seen twice* (i.e. don't-care) for the rest of the profile. This is because, in an access sequence for globals A, B, and C of ABCBCBCA... the second reference to A will restore A to the cache even if it was evicted by the earlier references to B and C. Therefore, any temporal affinity with the instance of A in the head CFG node for this path is now concluded. Conflict with the second A in the reference stream will be recorded when that second A reference is part of a head node for another path in the near future. The path analysis for a particular path continues as long as there are other globals in the head CFG node's data profile list that have not been marked as having been seen twice.

Because a CFP node can have two children, causing recursive calls to be made, it is also possible that paths through children nodes will eventually converge together again. For example, in a simple if-then-else control structure, each of the two branch directions eventually reaches the code that follows the if-then-else. It would be inaccurate to record affinity twice from that point onwards just because separate paths are being followed by separate recursive instances of the path analysis. To avoid this problem, each CFG node has a boolean field that indicates whether it has been visited. Before each path begins, all the visited flags are cleared. When a path analysis visits a node, the flags are set, and the second path to reach that node will terminate immediately to avoid double counting of affinity.

Notice that, by starting a path at each eligible CFG node, that the paths have a great deal of overlap. Only the possible evictions of the global variables in the head CFG node's data profile list are recorded in the affinity arrays, so there is no double counting. The overlapping scheme keeps the static path analysis code at a manageable level of complexity, and

it will be shown in Chapter 5 that the analysis time is still very fast (on the order of 30 seconds for fairly large benchmark programs).

3.4.1 Path Analysis without Complete Inlining

As mentioned above, the inlining of CFGs until only the expanded CFG for the `main()` function remains is a memory-intensive process, and it could be that the host system does not have enough virtual memory for completion of the inlining. In that case, path analysis must be performed in the presence of multiple remaining function CFGs, one being for the `main()` function. This situation has been encountered on some of the largest SPECint2000 benchmark codes, such as *176.gcc*, *253.perlbmk*, and *254.gap*. It is observed that there is usually enough time to inline all leaf functions, but the process of inlining the indirectly recursive chains of functions that remain after all leaf nodes are inlined does not reach completion.

A path analysis algorithm has been designed to address this situation effectively. A quick scan over the remaining CFGs reveals which function has the fewest remaining call sites (i.e. calls to other functions). This statistic is maintained in the call graph data structure as we inline; it is needed to determine which nodes are leaf nodes (i.e. a call site count of zero signifies a leaf node). Because the indirect recursion chain must be broken in some way, the chosen function is scanned and a list of globals and formal parameters accessed is compiled. The list includes the sum of the static execution frequency estimate fields from each CFG node wherein accesses occurred. The few call sites are ignored at this stage; this imprecision is the price paid for not being able to complete the inlining process. Then the remaining functions have similar lists of globals and formal parameters compiled for their accesses, in the same way, with the exception that not all call sites are ignored. If a call site specifies a function which has had its access list created, then the access list is *virtually inlined* at the call site; i.e. without any actual copying, the access list of the callee is added to the access list being built for the caller, with the execution frequency estimates of the callee's access list being multiplied by the frequency estimate of the calling CFG node. Matching of the actual parameter list in the calling CFG node to the formal parameter usage recorded in the callee's access list is performed in the same way as was described for inlining function CFGs.

The function with the fewest call sites that need to be ignored (i.e. fewest call sites to functions that do not have access lists yet) is chosen as the second function to have its access list created in the manner just described. Proceeding in this manner, the imprecision of ignoring call sites to break the chain of mutual recursion is minimized, until access lists have been created for all functions other than the `main()` function. At that point, interprocedural path analysis commences, differing from the path analysis previously described in that call sites remain in the `main()` function. The path analysis is confined to the `main()` function. When a call site is encountered, the access list for the callee is processed as if it existed in the calling CFG node within the `main()` function.

To see how this approach compares to path analysis when inlining was able to run to completion, consider a very simple example. The `main()` function calls a subroutine which contain a branch followed by two possible paths. If the one path is taken from the branch, global variables C and D are accessed. If the other path is taken from the branch, global variable E and F are accessed. The `main()` function accesses global variables A and B just prior to the call site. In ordinary (i.e. after complete inlining) path analysis, the access sequences ABCD and ABEF would be recorded, meaning that A and B would each be recorded as having conflict affinity with C, D, E, and F. In the access list method of path analysis, an access list containing CDEF, with execution frequency weights for each, is associated with the call site. Variables A and B thus continue to have conflict affinity recorded in the same manner as before.

A couple of hurdles still remain in order to make interprocedural access-list analysis accurate. Ordinary path analysis starts all paths in CFG nodes that are contained within loops, so that the execution frequency estimate is significant and the path does not terminate shortly after its beginning. A CFG node within a function that could not be inlined due to time and space constraints might appear to fall outside of any loops, but the function itself might actually be called from within loops in callers. In order to have a realistic execution frequency estimate within a function other than the `main()` function, a call frequency estimate field is maintained for each function and increased by the frequency estimate of calling CFG nodes whenever these nodes are encountered in a pass through all CFGs just prior to computing the access lists. This gives the path analysis implementation a multiplier for the frequency estimates in each CFG node in the function. If the multiplier

is greater than one, and at least one calling CFG node was within a loop, then the CFG within the function can be treated as if it is all within a loop. No paths will terminate within the function due to the execution frequency threshold, and paths are permitted to start within the function, because each CFG node within the function is significant.

The second potential problem with the accuracy of interprocedural access-list analysis in the cache capacity cutoff for paths. Recall our example above, in which a function accesses global variables C and D on one path, and E and F on another path. The access list includes all four of these globals, but they cannot all be accessed in a single path. When we compare the accumulated size of globals seen on a path with the capacity cutoff value (currently twice the cache size), we do not want to use the size of the union of all paths through a function, as this will lead to premature termination of paths during analysis. Instead, all paths from the entry to the exit of the function are followed in a preliminary step before the access lists are built. As the size of each path is accumulated, the probability of each path is multiplied by the final size of each path to produce a weighted average cumulative size of the globals accessed in the function. For example, because a two-way branch is assigned a probability of 0.5 for each branch direction, in our small example, the weighted average size of globals accessed would be half the sum of the sizes of C and D, plus half the sum of the sizes of E and F. This is far more accurate than summing the sizes of C, D, E, and F, as a naive use of the access list might otherwise imply.

The implementation of this design of interprocedural access-list analysis has not yet been completed and tested, although a portion of the code was written when the need for the design was first discovered. When benchmark results are given in Chapter 5, it will be seen that the common benchmark programs that are unable to be inlined to completion are also unsuitable for the present research due to intensive use of heap-allocated data, among other factors. Thus, finishing the coding and testing of this portion of the path analysis design was not necessary to complete this research, and is scheduled for future extensions to the present work.

3.4.2 Result of Path Analysis

At the conclusion of the path analysis for all eligible nodes (i.e. all CFG nodes with sufficient execution frequency estimates), temporal affinity arrays have been produced for

each level of cache (currently two levels, but easily expandable for future target machines with more levels of cache). The global data placement algorithms described in the next chapter have, in the affinity arrays for each level of cache, all of the data needed to drive the rest of the placement optimizations.

Chapter 4

Data Placement

A number of different objectives must be balanced against one another in the design of the data placement optimizations. Ideally, the optimizations would lead to simultaneous reductions in compulsory and conflict misses in each level of the cache hierarchy, as well as reductions in TLB (translation look-aside buffer) misses, addressing costs for global and local variables, and bus cycles. Of course, the ultimate goal of all such reductions is a reduction in execution time, although there will be other benefits (such as reduced energy consumption) that can be significant in certain applications.

This research breaks new ground in the innovative static analysis design presented in the previous chapter, in the design of a data placement framework that integrates the many concerns listed in the paragraph above, and in the automatic characterization of memory hierarchy parameters that are essential to the analysis and placement designs. The latter work will be explained in Chapter 6. In the present chapter, an integrated collection of data placement optimizations, addressing multiple performance objectives throughout the memory hierarchy, is presented. The high-level sequence of these optimizations can be seen in the following pseudocode:

Pack small global variables into cache lines.

Align all variables on the cache line size of the largest cache.

Partition all variables smaller than the highest level cache into partitions equal to or less than the size of that cache.

For each partition formed, partition further by the next smaller size of cache in the hierarchy, until the first cache level is reached.

Perform a local refinement across all partitions, in which globals are rearranged within their partitions to reduce cache conflicts.

Pad the arrays larger than the largest cache size to prevent their starting addresses from colliding in the first-level (and hence any other level) of cache.

The present target machines only have two cache levels, but it will be seen that the optimizations are designed to work on any number of cache levels. The optimizations are now described in order of their application, after explaining the building of the global symbol table used throughout all data placement and related optimizations.

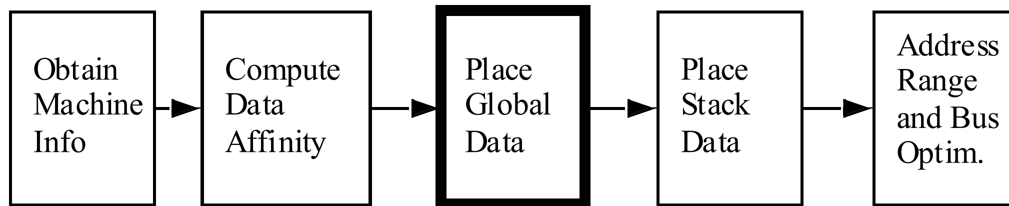
The Global Symbol Table

The front end of the compiler emits an intermediate code representation of the program. The *middleware* component of the compiler translates the intermediate code into *register transfer lists*, or RTLs (see [Davidson and Fraser 1980] for the design of RTLs and their use in *vpo*). Included in the intermediate code that is emitted by the front end are declaration lines for each global variable, formal parameter, and local variable. The declaration lines for formal parameters have already been used when building a list of formal parameters for each function, as described in Chapter 3, in order to match formal with actual parameters during static data analysis.

Recall from discussions in Chapter 3 that two passes are made through the RTL representation of the program whenever whole-program optimizations, such as data placement, are requested. In the first pass, a call graph is constructed. During this pass, static data analysis is performed up to the point at which the CFG for each function is created. At the conclusion of the pass, the CFGs are inlined and path analysis is performed. While processing the RTLs for each function in this first pass, the declaration lines for global variables are first seen. Each declaration line includes the name, unique internal numeric code, and size of the global. These attributes are entered into a global symbol table entry for the global, which is kept as a linked list. A synthesized attribute is also recorded: the index number into the affinity arrays for this global. As the globals are rearranged in the linked list to reflect data placement decisions, this number will remain constant.

4.1 Cache Line Packing

After the conclusion of the path analysis, the data placement optimizations are ready to begin. Recalling our diagram from Chapter 3:



Most of the rest of this chapter will describe the steps within the *Place Global Data* box, beginning with cache line packing.

Cache line packing attempts to reduce compulsory cache misses, as well as providing other benefits that will be described shortly. A *compulsory cache miss* is caused by the very first reference to a particular cache line in memory, where “particular” does *not* mean “modulo the cache size”. If the data space of a program is divided into cache-line-sized pieces for the level of cache being discussed, with each piece aligned by the cache line size, then the first reference within the program’s execution history to any data address that falls within a particular cache-line-sized piece causes a compulsory cache miss. Because cache lines are generally larger than individual array elements or scalar variables, programs will benefit from *spatial locality.* That is, if one data reference tends to be close in memory to the previous data reference, then many references will cause a cache hit because a previous cache miss caused an entire cache line to load, which included more than a single array item or scalar variable. If sequential accesses to array elements A[0], A[1], etc., are made by a program, and a cache line holds eight elements of array A, then only one access out of each eight accesses will cause a compulsory miss. The definition is given in detail here because some sources loosely (and mistakenly) define a compulsory cache miss as the kind of miss caused by the first access to a *data item* in a program’s history, in which case the first access to A[1] would cause a compulsory cache miss, regardless of it being in the same cache line as A[0].

In order to reduce compulsory cache misses, a data placement optimization needs to increase the number of times that a cache miss produces spatial locality benefits for ensu-

ing data accesses while the cache line that was loaded due to that miss remains in the cache. In pursuit of this goal, variables that are smaller than a cache line in size are more amenable to manipulation by a data placement optimization than are variables larger than a cache line. While an array could be smaller than a cache line (e.g. an array of eight characters is smaller than almost all cache lines in use today), the term scalar shall be used henceforth in this research as shorthand for all variables smaller than a cache line in whatever level of cache is being discussed. As the discussion of the data placement algorithm develops, it will be seen that the largest cache line size in the system is the primary point of reference. (Note that in all known cache hierarchies, each level of cache has at least as large a line size as the smaller cache level that is closer to the CPU.)

Arrays (i.e. arrays at least as big as a cache line) receive the benefits of spatial locality primarily by sequential access patterns, rather than by having code *stride* through the array with a gap between consecutive accesses of one or more elements. The loop transformations mentioned in the prior work discussion in Chapter 2 focus in part on the transformation of code to reduce the interaccess gap and enhance spatial locality. This research does not implement already published work in this respect. The access pattern within the array is not analyzed within the *vpo* infrastructure. The only spatial locality benefits that data placement confers upon arrays come from aligning the arrays so that their starting addresses are at zero, modulo the largest cache line size in the system. This ensures that each array occupies the minimum number of cache lines at all levels of cache. It further provides the benefit of spreading apart the first and second compulsory misses when the array is accessed sequentially, as compared to having the starting address of the array be nearer the end of the first cache line. This can be helpful on machine architectures with non-blocking caches which execute past cache misses until the result of the load that caused the miss is actually needed [Chen and Baer 1992]. It is best for such machines to have fewer outstanding cache misses at one time, and starting an array at the beginning of a cache line maximizes the temporal distance between the first two compulsory cache misses. It also simplifies later analyses, which build a map of the caches, with each cache line labelled by which variables occupy it. Having arrays occupy a whole number of cache lines, with padding bytes at the end of the final line if needed, simplifies the cache mapping while maximizing spatial locality.

Scalar variables can be much more effectively targeted for reductions in compulsory cache misses. The declarations of scalar variables are often scattered throughout the program text. For example, a scalar might be declared immediately before or after an array that it indexes. The default compiler data placement for global variables (in order of their declaration) will thus produce an interleaving of scalars and arrays (which, as a result, are not aligned on cache line boundaries). Using the temporal affinity that has been identified during static data analysis, scalars can be packed into cache lines together. The goals are to maximize the temporal affinity that exists within a cache line, to increase opportunities for load coalescing optimizations by placing scalars in adjacent positions if they have a high load affinity, and to reduce *pollution* of caches and virtual memory pages by increasing *cache line utilization*. Cache line utilization is the proportion of a cache line that is accessed before the line is evicted from the cache. If a cache miss brings in an entire cache line, only one element of which is accessed before that line is evicted, then all other elements of that cache line are said to be pollution in the line. Likewise, if a page is brought into virtual memory (and hence its corresponding TLB entry is brought into the TLB), but only a small portion of the page is accessed before it (or its TLB entry) must be swapped out, then the page suffered from pollution by the unused data items. With arrays declared in between scalars that have affinity, accessing one scalar and then the other could bring in two pages and two TLB entries for only a few bytes of data references.

When the optimization to reduce the addressing costs for globals is described, it will be seen that it is crucial to that optimization to have temporal affinity within the addressing range of a single base register. If scalars with mutual temporal affinity are separated by large arrays, it might not be possible for a single base register to span that distance. Cache line packing is thus an enabler of another optimization that follows data placement.

Because cache line packing can increase locality throughout the memory hierarchy, as well as enabling optimizations such as load coalescing and addressing cost reduction, and because it will simplify all further data placement analyses to have no data unit smaller than a cache line to analyze, cache line packing is the first data placement optimization described in this chapter. Different algorithms for packing cache lines are explained and compared in the subsections below. At the highest level of abstraction, the two categories of data-arranging algorithms for this problem are *bottom-up clustering*, in which the scalars

are grouped together until each group (approximately) fills a cache line, and *top-down partitioning*, in which the entire group of scalars are divided into subgroups which are then subdivided further until the groups have each become (approximately) the size of a cache line.

4.1.1 The Greedy Algorithm for Cache Line Packing

Given the temporal affinity array for each level of cache, the most direct greedy algorithm is to pair up the two scalars with the highest temporal affinity *density,* i.e. the highest mutual affinity divided by the combined size of the two scalars, with the restriction that the combined size cannot exceed the cache line size. The paired scalars are now marked in the global symbol table to indicate that they must remain paired together in all further data placement and cannot be separated. The paired scalars are now considered to be a single scalar henceforth. The pairing continues until all scalars with any mutual affinity have been paired. If any scalars remain, they are packed into cache lines that have not been completely filled yet, and then all lines are padded out with empty bytes to the full cache line size. Details for this implementation are given under the headings that follow.

Multilevel Cache Hierarchy Issues

A typical cache hierarchy can have several important cache line (a.k.a *block*) sizes. For example, in the Sun UltraSparc-IIi machines that have been used primarily for this research, the first-level data cache has 32-byte blocks with 16-byte subblocks. This means that a cache read miss causes 16 bytes to be fetched from the second-level cache (write misses cause no fetches in the first-level, write-through caches on these machines). The other 16-byte subblock within the 32-byte block is not yet fetched, but the old data in it is marked as invalid. Subblocking is a scheme to reduce cache bus traffic in the presence of cache pollution; if the second subblock is never referenced, no time was wasted in loading it. On the other hand, if it is almost always referenced, then it could have been loading “in the background” while execution continued. The second-level cache has 64-byte lines with no subblocking, and write misses cause old cache lines to be evicted and the referenced cache line to be loaded from main memory.

In the presence of such a cache hierarchy, it is desirable to maximize temporal local-

ity of scalars within each 64-byte second-level cache line, and also within each 32-byte first-level cache line, and also within each 16-byte first-level cache subblock. This is in fact accomplished to a large degree by the greedy algorithm or any other algorithm that performs a bottom-up clustering of the scalars. In each such algorithm, the affinity is greatest between scalars that were first paired together, next between members of pairs that got paired in the next stage, etc., so that affinity within a 16-byte subblock is greater than affinity across subblocks, for example. Therefore, all bottom-up clustering approaches are well suited to lay the foundation for a later load coalescing optimization.

Interactions with Cache Miss Forwarding

It is common in modern cache systems to employ *cache miss forwarding* circuitry in the hardware. Suppose that the running program requests a certain data item whose address falls towards then end of its cache line, and that the request causes a cache miss. It is desirable to minimize the delay in providing the requested data item. A simple cache miss handler circuit would load the entire cache line, and then provide the requested data item from the loaded cache line to the destination register in the CPU. With a little more hardware and design effort, the requested item will be streamed through to the CPU register as soon as it is available, while the rest of the cache line is still loading. This is possible because the first-level-cache-to-CPU bus is separate from the first-level-to-second-level-cache bus. Notice that the delay in providing the requested item has been reduced on average from a fixed amount of time to an amount proportional to the distance of the requested item from the front of the cache line. With greater complexity in hardware, the cache line fill can start with the requested data item and wrap around, treating the cache line as a circular buffer to be filled, so that the requested item can be forwarded to the CPU as early as possible while the remainder of the cache line fill completes.

One goal of this research is to make the data placement optimizations consider more aspects of modern computer architecture (and *micro-architecture*, i.e. the detailed implementation of an architecture that is hidden from the view of even an assembly language programmer) than previous efforts in this field. Another goal is to explore the effectiveness of compiler approaches to managing cache conflicts as opposed to the ever more complex hardware approaches that have been implemented in recent years such as the cache miss

forwarding techniques described above. If a machine implements the most advanced wrap-around cache line fill approach described above, then there would be little or no execution-time benefit to making cache misses fall closer to the beginning of cache lines on average. However, not all CPUs are designed with the same level of micro-architectural complexity. In certain embedded systems, considerations of die size, cost, and power consumption may dictate a simpler implementation than on server and workstation CPUs. More discussion of the interactions between micro-architectural design and this research is found in Chapter 7.

The static data analysis algorithms were tailored to enable ordering of data items within the cache line. Recall that if the static path analysis encounters an access to variable A before an access to variable B, where A and B have indices *i* and *j* into the affinity array, then only element (*i, j*) is increased in the affinity array. If B were accessed before A on a path, then only the (*j, i*) element in the affinity array is increased. By comparing these two affinity array values, it can be determined which variable usually precedes the other in access order. When the decision is made to pair A with B in any of the bottom-up clustering algorithms for cache line packing, the order of the two is decided once and for all based on the magnitude of their mutual values in the affinity array for the first-level cache. The greatest number of cache misses happen in the first-level cache, and the effect on the cache hierarchy filters down from that cache to the other levels, so optimal ordering is achieved within pairs by giving precedence to the first-level affinity array. As this pairing continues, the process reaches the point of pairing larger items (perhaps previously paired items, now treated as single items) and the result is to make it more likely that a second-level cache miss will occur due to a request for a data item in the first-level cache subblock that lies at the beginning of the second-level cache line, rather than in the later subblocks. Thus the data placement can serve to reduce the need for wrap-around cache miss forwarding in the interface between main memory and the second-level cache, just as it did for the interface between the levels of cache.

Effect of Cache Line Packing on the Affinity Arrays

As each pair of scalar variables is formed, the order within the pair is determined as noted above. At this point, each bottom-up clustering cache line packing algorithm will coalesce the affinity array entries for the rightmost scalar in the pair into the affinity array entries for

the leftmost scalar in the pair. That is, if scalar A has array index *i* and scalar B has array index *j*, and A and B were just paired with A leftmost and B rightmost, then row *j* in each affinity array is added to row *i*, with row *i* receiving the result and row *j* being set to all zeroes afterwards. This means that the two entries in an affinity array that formerly meant "A is accessed this many times before C" and "B is accessed this many times before C" are now interpreted to mean "The A-B pair is accessed this many times before C". Similarly, column *j* is added to column *i*, with column *i* receiving the result and column *j* being set to all zeroes afterwards. This means that "C being accessed before A", and "C being accessed before B", have been added together to mean "C being accessed before the A-B pair". After the completion of the cache line packing phase, no data item smaller than the second-level cache size is analyzed or placed.

4.1.2 A Graph Matching Algorithm for Cache Line Packing

The affinity arrays produced by static data analysis can be treated as a representation of a complete graph, in which each global variable is a node in the graph, and the sum of the two affinity array entries (*i, j*) and (*j, i*) are the weight of the undirected edge connecting nodes *i* and *j*. Several algorithms attempt to optimally match (i.e. pair) the nodes within a graph based on the weight of the edges between the nodes. The problem that best matches the goal of cache line packing by bottom-up clustering is the *nonbipartite weighted graph minimum weight perfect matching* problem. This is referred to simply as the nonbipartite weighted matching problem in Chapter 11 of the classic combinatorial optimization work by Papadimitriou and Steiglitz [Papadimitriou and Steiglitz 1998]. An optimal solution algorithm with $O(n^4)$ runtime, where *n* is the number of graph nodes, is given there. More efficient optimal algorithms with $O(n^3)$ runtime are known in the literature [Gabow 1973].

In a single sweep through the nodes of a complete weighted graph, in $O(n^3)$ time, the algorithm of Gabow will form pairs (matchings) of all nodes such that the sum of the edge weights used in the matching is maximized. In contrast, the greedy method outlined above traverses an $O(n^2)$ affinity array in order to make a single pairing. In the worst case, all variables in the program are scalars and $O(n)$ sweeps through the array are required to use the greedy method to completion, whereas $O(\log n)$ sweeps would be needed to apply the maximum nonbipartite weighted graph matching algorithm to completion. The bench-

mark programs used in this work generally have a value of n in the hundreds but less than 1000. Not all of these variables are scalars, and termination occurs when the pairing has reached the size of a cache line, not when every scalar in the program has been paired hierarchically into a single chain of scalars. Thus, the number of sweeps required for the graph matching approach to cache line packing is $O(\log b)$, where b is the block size (line size) of the level of cache targeted by the optimization. Given the 64-byte second-level cache line sizes of the UltraSparc-IIi machines used primarily in this research, no more than 6 sweeps would be required to pack all scalars into cache lines, regardless of the value of n. Thus, the time complexity would be $O(6n^3) = O(n^3)$. The constant factors of the implementations could be decisive for these values of n, so only dual implementation and timings can determine which is more efficient in practice.

The effectiveness criterion is the sum of mutual affinities that have been confined within cache lines, which should be maximized. This is equivalent to stating that the sum of mutual affinities that crosses cache line boundaries needs to be minimized. In all bottom-up clustering algorithms, the problem solved in a single sweep is well defined, but the relationship of that problem to the higher level problem of packing cache lines with maximum effectiveness is imperfect. This point will be discussed below in the context of graph partitioning. As far as the effectiveness of different bottom-up clustering algorithms when compared to each other, the suspicion is always that a greedy algorithm is simple, because it makes only local decisions, but globally suboptimal because it does not make global decisions. However, some greedy methods (e.g. greedy knapsack packing) approach optimality as the value of n increases, because the global effectiveness is not as sensitive to the final few local decisions, where the suboptimality of previous local decision is typically exposed. Again, only measurements will determine the relative effectiveness of the different algorithms proposed for cache line packing.

4.1.3 Cache Line Packing by Graph Partitioning

The top-down, global view of the cache line packing problem is: Given a group of scalars and their mutual affinities, form them into groups of size no larger than the cache line size such that the sum of affinities within cache lines is maximized. There is a graph problem that directly corresponds to this cache line packing problem: Given a weighted graph,

partition the graph into subgraphs, each of size (i.e. number of nodes) no greater than a certain limit, such that the sum of edge weights that cross subgraph boundaries is minimized.

There is a simple proof that maximizing the edge weights that do not cross subgraph boundaries after partitioning is the same problem as minimizing the edge weights that cross subgraph boundaries after partitioning. Given a graph with nodes, edges, and weights for each edge, let T denote the total sum of all edge weights, let I denote the total sum of internal edges weights (i.e. weights of edges that stay within a subgraph), and let E denote the total sum of external edges weights (i.e. weights of edges that cross subgraph boundaries). As every edge must cross or not cross subgraph boundaries after partitioning is complete, then $T = I + E$. None of the edge weights are changed by partitioning, so T is constant throughout partitioning; only I and E are variable with respect to the partitioning of the graph. Therefore, maximizing I is the same as maximizing $T-E$. With T constant, the expression $T-E$ is maximized by minimizing E. By similar reasoning, minimizing E maximizes I. Bottom-up clustering attempts to maximize I, while graph partitioning is defined in terms of minimizing E, which is an equivalent problem.

There is some reason to believe that graph partitioning is superior to bottom-up clustering for maximizing temporal affinity within a given address range, such as the size of a cache line. In early efforts to reorder data to minimize page faults in virtual memory systems, it was noted that a bottom-up clustering approach tended to do a good job of clustering data together into pages until the last page, which ended up being a hodgepodge of data items that had little temporal affinity for one another [Kernighan and Lin 1970]. Kernighan and Lin observed that this seemed to be due to the inherently greedy nature of the local decisions made in a bottom-up algorithm. If four data items from a certain set of five data items with high affinity for one another would fill up a page, then the (unspecified) bottom-up clustering method that was used would omit the item with the lowest sum of affinities for the other four items. However, the benefit on that page might be very small as compared to omitting the item with the second lowest sum of affinities, whereas there might be a large difference in another page depending on which of those two items were available. It should be noted that the clustering method that was measured does not sound exactly like either of the two clustering methods described in this research. Apparently,

clustering was done until the first page was filled, then more clustering filled a second page, etc. In the greedy clustering method implemented in this research, the best pair is formed first, then the second best pair is formed. At this point, note that no cache line is necessarily filled yet; the two pairs that have been formed might end up in different cache lines in the final placement. Likewise, the graph matching approach does not fill cache lines one at a time; it forms a set of pairs, which are then paired recursively.

Cache line packing can be implemented using graph partitioning. Kistler and Franz did exactly this in the limited context of rearranging fields within heap-allocated data structures in order to pack cache lines more efficiently [Kistler and Franz 2000]. Their graph partitioning code required that all nodes be of unit size, so they split each n-byte data item (i.e. field within a structure) into n 1-byte items, with each having infinite affinity with the other n-1 items in order to force them to remain in the same final partition. Enough dummy nodes of size 1 byte each, with no affinity for any other node, are added to the graph to make the total size equal to a power of two bytes. Then graph bipartitioning can be applied recursively until the cache line size is reached for all subgraphs.

The graph partitioning code used in this research is *Chaco* [Hendrickson and Leland 1995a], a freely available graph partitioning code that is linked as a separate library into *vpo*. *Chaco* is used in the significant graph partitioning steps of the main data placement code that follows cache line packing, as will be described below. *Chaco* could be used for cache line packing, but there are two difficulties with doing so, one of which is true for all graph partitioning codes and one of which might not be:

1. An extensive post-processing step would have to be added after the graph partitioning is complete in order to order data items within the cache line to properly place the data for load coalescing and to move compulsory cache misses closer to the beginning of each subblock. This placement within the cache line was part of the design of the bottom-up clustering algorithms, and some sort of clustering code would have to be applied to each cache line.
2. *Chaco* does not always create subgraphs of exactly the same size. It will be shown below that, when *Chaco* is called upon to split 32KB of data into two pieces, it is preferable to produce a 17KB subgraph and a 15KB subgraph, rather than two 16KB subgraphs, if the external edge weights are thereby reduced. However, if some subgraphs come out larger than a cache line when *Chaco* is used for cache line packing, then it is unclear how effective the solution will be after a postprocessing stage cleans up the partitioning, and also unclear how to optimally design such a postprocessing stage.

Because of these complications, cache line packing by graph partitioning using *Chaco* has not yet been implemented. It will be implemented, and its effectiveness compared to the bottom-up clustering approaches, in follow-on work.

4.1.4 Other Methods for Cache Line Packing

It would be possible to implement cache line packing by other means. For example, a bottom-up clustering approach such as the one apparently described by Kernighan in his page placement research, in which cache lines are packed one at a time, could be implemented and compared to the other approaches. This could give insight into whether the supposed inferiority of bottom-up clustering to graph partitioning methods is actually an artifact of this particular approach, as currently suspected. Other graph partitioning codes could be used and compared to *Chaco*. For the present research, the bottom-up clustering implementation seems effective enough to indicate that only marginal improvements are likely from any other method.

4.2 Data Placement by Graph Partitioning

One unique contribution of this research is the formulation of the overall data placement problem (after the completion of cache line packing) as a hierarchical graph partitioning problem. If all global variables in the program are represented as nodes in a graph, with the edge weights in the graph representing the temporal affinity between the nodes join by the edge, then partitioning a subgraph consisting of all global variables that are less than the size of the second-level cache into partitions which are each the same size as the second-level cache, such that the external edge weights are minimized (and thus the internal edge weights are maximized) can serve as the first step towards reducing conflict in the second-level cache. This problem is the well known graph partitioning problem, an NP-complete problem (see p. 209 of [Garey and Johnson 1979]) of wide interest in various applications. The *Chaco* graph partitioning package mentioned above was developed in the application area of partitioning tasks across multiple processors to reduce message passing costs.

The interface to *Chaco* requires an adjacency list representation of the graph. *Chaco* permits nodes to have a non-unit size, so that it is not necessary to split an *n*-byte data item

into *n* 1-byte items, as was the case with the graph partitioning code used by Kistler and Franz and discussed in Section 4.1.3 above. It is best to give maximum flexibility to *Chaco* by providing dummy graph nodes, with no connectivity to other nodes, such that the total node size is a power of two.

Using *Chaco* to partition a program's global data follows the steps listed below:

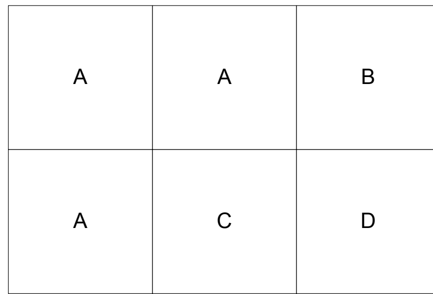
1. The affinity arrays are scaled so that the maximum edge weight presented to *Chaco* will be 10000. This limit is explained below.
2. The global symbol table is traversed. For each global, using its index into the affinity arrays, a list of adjacent nodes (i.e. nodes having a positive affinity value in the second-level affinity array) is formed for all global variables smaller than the second-level cache, along with a list of edge weights for the edges to those adjacent nodes. Note that globals that are at least as large as the second-level cache are excluded; they conflict with every other variable in all levels of cache regardless of their placement. If the total size of the variables that are smaller than the second-level cache is not enough to exceed the size of the second-level cache, then conflict in that cache cannot be reduced by graph partitioning and the processing skips down to step 7, where all variables smaller than the first-level cache are treated as if this stage had put them in one second-level cache partition.
3. As *Chaco* numbers the nodes by their position in the adjacency list, and not all nodes are present in the adjacency list (i.e. some nodes have no temporal affinity with any other nodes, perhaps being accessed only outside of all loops), a mapping of *Chaco* node number to *vpo* global number is created.
4. Dummy nodes are appended to the end of the adjacency list to increase its size to the next power of two, in bytes. Each dummy node has the same size as the second-level cache line size and has no edges.
5. *Chaco* is called, passing it the adjacency list, and receiving back an assignment of global variables to partition numbers.
6. The assignment from *Chaco* is related to the global symbol table entries using the mapping created in step 3 above. Each global is marked with the assigned partition number, multiplied by the number of first-level cache partitions that can fit in the second-level cache. Thus, if the second-level cache is 256KB and the first level cache is 16KB, the ratio is 16. If *Chaco* assigned partition numbers 0, 1, 2, and 3 to the globals, then the globals will actually be marked with partition numbers 0, 16, 32, and 48. This permits each partition to be partitioned for the next smaller level of cache and marked with consecutive numbers.
7. For each partition returned from *Chaco*, an adjacency list is created for those variables that are smaller than the first-level cache size, only if such a list would have a total size that exceeds the first-level cache. For example, suppose that a 256KB partition has been formed, using globals of size 145KB, 100KB, and various small globals with a collective size of 11KB. The 11KB of small globals in this partition cannot be partitioned further for a 16KB first-level cache, so we move on to the processing of the next 256KB partition.
8. After *Chaco* assigns zero-based partition numbers for the small globals, the assigned partition number is added to the base partition number for the original larger partition, and the globals are marked accordingly in the global symbol table. For example, if the first *Chaco* partitioning

is as described in step 6, producing partitions 0, 16, 32, and 48, then the partition assignments of 0, 1, 2, etc., returned by *Chaco* when it is given the smaller globals from within partition 16, are now translated into partition numbers 16, 17, 18, etc.

9. *Chaco* processing is complete and the final partitions are ready for the local refinement stage that will be described below.

It is appropriate at this point to explain some details that were omitted from the above list, and to explain the decisions underlying the design at some of these points. The scaling of affinity arrays in step one is based on recommendations from the *Chaco* user manual [Hendrickson and Leland 1995a]. *Chaco* takes the floating-point edge weights and converts them internally to integers. Page 38 of the *Chaco* user manual suggests that the weights be scaled so that they “are neither very small nor very big.” Therefore, if the largest edge weight will be smaller than 10000, we scale up in order to not pass very small weights., but it is more common to have to scale down the get the largest edge weight down to 10000. The number 10000 was derived by trial and error, observing the warning messages from *Chaco* with different values.

In steps two and seven above, the size cutoff for global variables to pass to *Chaco* was set at the respective cache sizes. In the early stages of this research, the cutoff was set at the *way size* of each cache, i.e. the cache size divided by the associativity. For the direct-mapped caches used on the UltraSparc-IIi machines, there is no difference here. For other machines, the distinction could be important. At first, it seems that any variable that is as large as the way size of a cache will conflict with every other variable in that level of the cache, and thus data placement will be ineffective with respect to such a variable and it could be omitted from this stage of graph partitioning. However, consider the conflict between variables A and B in the two different placements in figure 4.1 below:



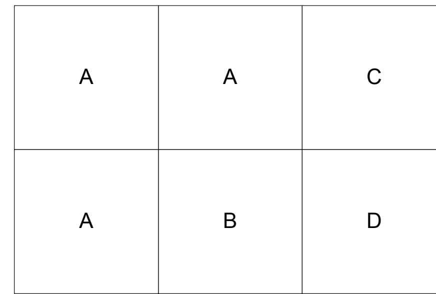


Figure 4.1(a)

Figure 4.1(b)

The height of the columns represents the way size of a cache. Variables occupying the same row map to the same cache line address and therefore are in conflict with one another. The variable sizes are conveniently chosen to be 1.5 times the cache way size for A, and 0.5 times the cache way size for B, C, and D. It can be assumed in this example that the cache is either direct mapped or two-way associative. Global variable A conflicts with all other variables in this level of cache. However, suppose that the affinity array for this cache level records far more affinity between A and B than it records between A and C or between A and D. Minimizing conflict between A and B is thus of primary concern. In Figure 4.1(a), B conflicts with the first and final thirds of the data space allocated to A, while in Figure 4.1(b), B conflicts only with the middle third of A. The latter placement is more desirable, yet omitting A from this stage of placement on the grounds that it is bigger than the way size and conflicts with everything would hide the different degrees of cache conflict that are caused by the different placements of B, C, and D.

Creating dummy nodes, and then ignoring their partition assignments from *Chaco*, leaves partitions that are smaller than the targeted cache size. After stripping the dummy nodes from a pair of 16KB partitions, there might be 14KB and 12KB remaining, for example. Because *Chaco* gains effectiveness from the flexibility of being able to place dummy nodes in any partition, it is preferable to permit the partitions (minus dummy nodes) to be of somewhat different sizes. Note also that if a 14KB partition is immediately followed in memory by a 12KB partition, as in our example, then the first 2KB of the 12KB partition will not have any conflict at all with the 14KB partition, and the last 4KB of the 14KB partition will not have any conflict at all with the 12KB partition, as in Figure 4.2 below.

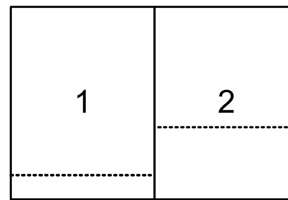


Figure 4.2

In the figure, partition 1 consumes 14KB of the 16KB cache size, so partition 2 consumes the final 2KB before wrapping around to the top of the cache and continuing for 10KB. Note that it might seem desirable to place variables within partition 2 such that the variables with the most affinity with partition 1 would fall (at least in part) in the first 2KB of partition 2, where there is no conflict with partition 1. Likewise, it would seem desirable to place the variables within partition 1 that have the most affinity with partition 2 at the end of partition 1, where the last 4KB has no conflict with partition 2. These considerations are handled in the local refinement of the placement, described in section 4.4 below.

4.3 Fine Tuning the Graph Partitioning

Chaco has a number of parameters that can be specified at run time to control the graph partitioning. Several of these parameters are significant with respect to the optimizations performed in this research. We discuss each of the significant parameters below.

Chaco implements several different graph partitioning algorithms. Three fast and simple methods are provided primarily for the sake of making comparisons. In the *linear* scheme, the partitions are filled by using the nodes in the order in which they were given. In the *random* scheme, a random number generator is used to assign nodes to partitions, with the probabilities weighted according to remaining space in each partition in order to balance the size of the final partitions. In the *scattered* scheme, nodes are assigned in round-robin fashion, with the first node going to the first partition, second node to the second partition, etc. After each partition has one item, the remaining items are placed one at a time in whatever partition has the smallest current size.

Three more complex (and more effective) partitioning algorithms are provided. The *inertial* method requires geometric information about the position of each node in one, tow, or three dimensions, and is suitable for applications dealing with a known geometric component [Williams 1991]. It is not suitable for partitioning global variables, which have no geometric position. *Spectral* partitioning methods use a complex mathematical relationship between the eigenvectors of a matrix representation of the graph, and the desired partitions of the graph, to bisect the graph. The spectral method described by the authors of *Chaco*, and used within *Chaco*, can also quadrisect or octasect the graph [Hendrickson and Leland 1995b]. In cases where the graph must be partitioned into more than two pieces, this could be advantageous, as will be discussed shortly. Finally, the *multi-level Kernighan-Lin* method of graph partitioning finds small groups of nodes with high edge weights within the group and clusters then into supernodes [Hendrickson and Leland 1995c]. After the graph is successively coarsened by this process and has passed under a certain size threshold, the coarsened graph is partitioned by a spectral method, and then the results of the partitioning are propagated back up to the original graph. At each coarsening step, the Kernighan-Lin algorithm is invoked as a local refinement method. (The Kernighan-Lin algorithm was designed to partition graphs by choosing an initial partition, perhaps at random, and then applying a specific local refinement technique until no further improvement could be made. As a result, it is well suited to being used as a local refinement stage for a different graph partitioning algorithm.) The authors of *Chaco* have experienced the best results with the multi-level Kernighan-Lin algorithm, which produces better partitions than the inertial and spectral methods on most, but not all, test graphs. It also runs faster than any spectral method tested, though not as fast as the inertial method.

Chaco also permits a local refinement to be performed at the conclusion of partitioning. Currently, the Kernighan-Lin algorithm is the only local refinement algorithm implemented. Note that using Kernighan-Lin refinement after multi-level Kernighan-Lin partitioning is not redundant; the uses of Kernighan-Lin refinement during the coarsening stages does not prevent further improvement by Kernighan-Lin refinement in the final partitioning after it is uncoarsened. For this research, the multi-level Kernighan-Lin partitioning, followed by Kernighan-Lin local refinement, will be the primary selection from these choices, based on the recommendations of the authors of *Chaco*. If time permits, compar-

ative measurements will be made with spectral partitioning followed by Kernighan-Lin local refinement.

A couple of aspects of the Kernighan-Lin refinement can be modified within *Chaco* in a parameters file, before the *Chaco* code is compiled and linked into a library for use by *vpo*. The REFINE_PARTITION parameter requests an extensive pairwise Kernighan-Lin refinement over all pairs of partitions in the final graph. The normal application of Kernighan-Lin refinement has been adapted to operate quickly over many partitions at once, moving nodes from one partition to another to improve the global metrics of effectiveness (these metrics are discussed further in the next paragraph). Pairwise refinement is more effective but more expensive in partitioning time. Because of the time expense, REFINE_PARTITION is set to zero by default. A nonzero value requests a certain number of refinement cycles over all pairs of partitions. A value of one will be used in this research as the default and then compared to other settings.

A second control over Kernighan-Lin refinement comes from the fact that some applications of graph partitioning (e.g. reducing inter-processor message passing in a multiprocessor system) not only desire a reduction in edge weights that cross partition boundaries, but also desire a reduction in the product of external edge weight times the distance between nodes in a given geometric interpretation of the graph. For example, it could be desirable to not only maximize the number of intertask messages that remain in a single node of a given mesh or hypercube architecture, but also to reduce the number of hops that a message must traverse to reach its destination. These two goals can conflict; to resolve the conflict when making local refinement decisions in the Kernighan-Lin algorithm, the effectiveness of the partitioning must be measured by selecting just one of these two goals. The default is make the hops metric primary, by setting the KL_METRIC to 2, which is not necessarily suited to the data placement application. In data placement, the external edge weights are an indicator of potential cache conflicts. If the only optimization related to the placement of data were the reduction of cache conflict misses, then the only concern would be reducing the *cut* (i.e. the external edge weights) of the final graph, by setting the KL_METRIC to 1, and there would be no attention given to reducing hops across partitions.

However, there are some other considerations besides reducing the cut edge weights

and hence the potential cache conflict. Before discussing these, we note that there is even a cache conflict concern that correlates to the hop metric. Recall Figure 4.2 above. It was noted in the discussion of that diagram that it will be desirable to utilize the 2KB of address space within partition 2 that has no conflict with partition 1, and likewise the 4KB of address space within partition 1 that has no conflict with partition 2. If partitions 1 and 2 have more conflict with each other than with any other partitions, then we can see that it is ideal for them to be adjacent to each other in the final placement, as this maximizes the conflict-free address spaces. If these partitions were separated by other partitions, it would be possible in the worst case for all 12KB of partition 2 to conflict with partition 1, and only 2KB of partition 1 would not conflict with partition 2 in the worst case. Distances between these partitions greater than 1 (i.e. adjacency) correspond to a hop metric in a one-dimensional mesh architecture.

The considerations, other than cache misses, that affect the choice of hop metric or cut edge weight metric for Kernighan-Lin refinement are addressing ranges and page locality. The typical linker places the data section of the compiled program at the start of a page boundary. Most machines use a page size that is smaller than the first-level cache size; common page sizes are 4KB and 8KB. Thus, it is likely that the first 2KB of partition 2 in Figure 4.2 will fall within the same page as the last 2KB (for 4KB pages) or 6KB (for 8KB pages) of partition 1. In general, adjacent partitions will share some pages, and page locality will be affected by the temporal affinity (or lack of it) between adjacent partitions. Likewise, as base registers are used to span several global variables and reduce the addressing costs within functions that access more than one global variable, the address space that is addressable using an immediate offset from that base register can be made to cross partition boundaries. If adjacent partitions have little affinity, this will not be of much use, whereas it could increase the opportunity for the addressing range optimization of adjacent partitions have high affinity.

In addition to the KL_METRIC parameter, the affinity between adjacent partitions in the final placement is affected by the mapping of the partitions to processors. Because *Chaco* was designed for parallel processing applications, such as mapping either tasks or data to processors, the user is required to specify the topology of the architecture being modeled. The choices are one, two, and three dimensional meshes, and hypercubes of any

number of dimensions. The natural choice for data placement is the one-dimensional mesh, as the partitions of global variables will be laid out in a linear address space. This is the only choice used and measured in this research. *Chaco* will, by default, try to assign partition numbers to the partitions it creates so that locality is maximized on the chosen architecture. However, it is possible to spend some more time and perform swappings of adjacent partitions that will improve the locality. If the REFINE_MAP parameter is set to TRUE when the *Chaco* code is compiled, then such swappings will be performed as a final step within *Chaco*, until no further improvement is possible. The default setting is FALSE, but the TRUE setting will be the default in this research, as all placement-related considerations (cache conflicts, page locality, and addressing range optimization) favor higher affinity among adjacent partitions.

The only prior use of graph partitioning in any form of data placement, as mentioned previously, was the rearrangement of fields within structures by the run-time compiler of Kistler and Franz. They used a standard graph partitioning code that was similar to Kernighan-Lin, but which was significantly faster due to data structure improvements. This code bisects a graph. If a graph needs to be divided into 16 pieces, then it is first bisected, then each piece bisected again, etc., until the partitions have reached the desired size. There is a lack of optimality in this process, however. It could be that a certain subset of the graph nodes are placed into a single partition during the first bisection, but they will not end up in the same final partition after further bisections (perhaps due to size constraints, for example). Intuitively, it is apparent that a local optimization decision has been made (the initial bisection) with a lack of some of the information that will be available to later optimization decisions, which is the recurrent algorithmic problem of *locally optimal* versus *globally optimal*. If the initial decision could have been informed that a certain group of eight nodes were to later be divided down into pairs of nodes, it might not be important for all eight to be in the same initial partition. The authors of *Chaco* developed multilevel versions of both spectral partitioning and multilevel Kernighan-Lin partitioning, in which it is possible to quadrisect or octasect a graph in a single stage. Both of these algorithms in *Chaco* are implemented in this manner. Experiments by the authors have confirmed their greater effectiveness as compared to recursive bisection. Explosive growth in memory requirements caused the limit to be set at octasection, although the technique is extensible to higher

degrees. In this research, the highest degree of partitioning possible is used on each call to *Chaco*.

A final concern in the operation of *Chaco* is related to the hop metric. When a graph bisection (or quadrisection or octasection) operation has decided that nodes A and B are to be placed in two separate partitions, the (hopefully small) edge weight between A and B is never considered again. The partitions containing A and B are then partitioned further, and it is possible that A and B and moving farther apart, in terms of the final partition's hop metric, as the partitioning stages progress. The same could be true of A and C, where C was initially placed in the same partition as B, but has now been separated from B and is drifting even further away from A. The REFINE_MAP parameter only controls a refinement after partitioning is complete, at which point the nodes that have the highest cut edge weights to A might have been scattered into many partitions, not all of which can be placed adjacent to the partition containing A. Thus, the hop costs have been affected by local decisions that lacked the global view needed.

However, the global information (i.e. the list of cut edge weights) did not disappear; it was simply ignored. Dunlop and Kernighan noticed this problem in the context of circuit placement, where allowing the nodes with affinity for A in our example to drift apart meant that long wires had to connect circuit elements that had not quite enough affinity to end up in the same final partition [Dunlop and Kernighan 1985]. In response to this problem, they developed a modification to Kernighan-Lin partitioning called *terminal propagation*. Information about the outgoing edges from a partition is passed along in the partitioning process. When the partition is further partitioned, the nodes with a common affinity for a distant node are more likely to be kept in the same subpartition rather than separated from each other. Of course, if there was any danger of their separation, it was because partitioning with cut edge weights as the only criterion would have reduced the cut edge weights by separating them. Thus, there is a trade-off between reducing cuts and reducing hops in the final partition. This trade-off is controlled by the CUT_TO_HOP_COST parameter, which is active whenever the TERM_PROP parameter is set to TRUE. The default value for CUT_TO_HOP_COST is 1, meaning that the ratio of the weightings given to the cut metric and the hop metric is 1.0, which gives the two metrics equal weight. Based on an intuition that the present research places a higher priority on cache conflict reduction than on adja-

cent partition affinity (in part because *Chaco* makes some effort towards the latter anyway, and the REFINE_MAP parameter is forcing even more effort after partitioning), the TERM_PROP parameter is set to TRUE by default, with the CUT_TO_HOP_COST parameter increased to 5. Different values will be measured and compared for effectiveness. The authors of *Chaco* have applied terminal propagation to both the multilevel Kernighan-Lin and the spectral algorithms in *Chaco*. Their experience is that terminal propagation enhances the partition quality quite a bit more consistently in the former than in the latter, as the combination of spectral methods with terminal propagation in quite mathematically complicated and sensitive, while only minor modifications are needed in a Kernighan-Lin (KL) algorithm.

A summary of the default controls over *Chaco*'s operation chosen for use in this research, with rationale for each:

- Multilevel Kernighan-Lin partitioning with KL local refinement, based on the recommendations of the authors of *Chaco*.
- Selection of the maximum degree of partitioning at each stage (e.g. octasection if possible) in order to reduce stages and reduce the suboptimality introduced by recursive bisection.
- Selection of terminal propagation, with trade-offs biased towards cut edge weights via the CUT_TO_HOP_COST parameter, in order to derive some adjacent partition affinity benefits for a variety of optimizations while still putting primary emphasis on reducing potential cache conflicts.
- Setting REFINE_PARTITION to TRUE to force a pairwise KL refinement to reduce cut edge weights in the final partitioning.
- Selecting a one-dimensional mesh architecture as the only logical model of a linear address space in memory.
- Setting REFINE_MAP to TRUE to increase adjacent partition affinity at no cost to cut edge weight reduction.

4.4 Local Refinement of the Data Placement Solution

Graph partitioning has helped to minimize potential cache conflicts by maximizing the affinity that lies within partitions that are less than the size of each cache, and minimizing affinity between such partitions. However, this does not complete the process of minimizing cache conflicts. Referring again to Figure 4.2 above, we noted earlier that it is desirable to minimize the conflict between partitions 1 and 2 by making optimal use of the address ranges that do not overlap both partitions; these totaled 6KB in that example. It is also the case that the 10KB of overlapping address range can be used with more or less conflict depending on the arrangement of global variables within the partitions.

The situation is complicated by the fact that a real application program will be likely to have enough global data to fill many partitions of the size of the first-level cache, even when we consider only the variables that are each smaller than the cache. The same is true for the second-level cache. Statistics on the global variable sizes for benchmark programs will be given in Chapter 5. If greedy local optimization decisions are made with respect to partitions 1 and 2 only, then it could be that such decisions constrain the placement of variables within later partitions and are not globally optimal. However, the problem of globally rearranging data within each partition to minimize conflict seems exponentially complex. This problem certainly seems at least as complex as the 2-D bin packing problem, for example, which is a provably NP-hard problem. In that problem, a collection of rectangular two-dimensional geometric objects must be packed into a two dimensional bin in order to minimize the height of the bin. An example is shown in Figure 4.3 below:

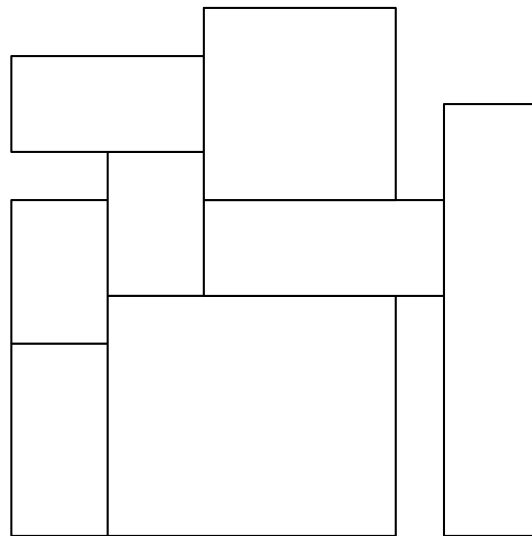


Figure 4.3
2-D bin packing

Data placement to minimize conflict could be considered to be somewhat analogous to the 2-D bin packing problem [Coppersmith and Raghavan 1989], with the height of objects representing their conflict metric with other objects above and below them in the bin, and their width representing the number of cache lines they occupy. A large difference in the problems is that the size of the objects would therefore change depending on where they are placed, and even depending on the later placement of an object above them. In addition, when objects wrap around the end of the cache and resume at cache line number zero, there would have to be a constraint linking the two pieces together so that they are forced to move in tandem when refinements to the packing are made. With the 2-D bin packing problem, without these complications, already being NP-hard [Seiden and van Stee 2002], it seems obvious that heuristic approximations, not optimal solutions, will have to be applied to the problem of rearranging the data placement within each partition to minimize conflicts.

Therefore, a local refinement of the placement within each partition must be designed. It is desirable that this local refinement will make decisions within each partition that reduce conflict globally, not just with respect to adjacent partitions. It is also desirable

that the refinement be able to proceed iteratively over all partitions until the improvement falls below some threshold, at which point it is not worth further compilation time to make more iterations. Finally, while it is not possible to enumerate all arrangements within all partitions, it is desirable to methodically search the solution space of arrangements so that local optima can be escaped with some degree of confidence.

In order to measure the global effects of rearrangements within a partition, a *cache map* data structure was designed. The cache map is an array, each of whose entries represents a cache line in the second-level cache. After cache line packing, each global variable is subsumed into a unit that is at least as big as a second-level cache line. The global symbol table (which is held in a linked list) is sorted by partition assignment number after *Chaco* processing is complete. This list is traversed, and the cache map array entries are filled in with pointers to the global variables that occupy each cache line. For a packed cache line of scalars, only the first scalar in the line is referenced; all others can be found from the links that start with the first scalar in the line. For an array that occupies many cache lines, each entry in the cache map contains the same pointer to that array variable in the global symbol table.

A function to compute the *conflict metric* for the current cache mapping uses the cache size for the level of cache whose conflict metric is being computed, along with the associativity of that cache, to determine which cache map entries wrap around the cache way size and conflict with each other (i.e. fall on the same cache line number). For any cache line number in which the number of globals that map to it, and which have affinity for at least one other element that also maps to that line number, is less than or equal to the associativity of the cache, a conflict metric of zero is recorded; the affinity these variables have for one another is irrelevant if they can all fit in the cache, or if the associativity of the cache can avoid conflict misses. For each cache line that has more global variables with mutual affinity mapped to it than the associativity of the cache, a conflict metric is computed as follows, where each wrap-around of the cache way size is referred to as a *column*:

```
for column1 = 1 to maxcolumn - 1
   let i = index into affinity arrays of global in column1
   for column2 = column1 + 1 to maxcolumn
      let j = index into affinity arrays of global in column2
      if i == j continue to next value of column2
```

```
        compute proportion_i and proportion_j
        set proportion to the maximum of proportion_i and proportion_j
        increment cumulative conflict metric by proportion
           multiplied by the affinity array[i, j] entry
        increment cumulative conflict metric by proportion
           multiplied by the affinity array[j, i] entry
    endfor
endfor
```

The proportion factors are computed based on the following reasoning. Recall the lengthy discussion in the path analysis section of the previous chapter, in which the affinity array values were scaled, based on variable sizes and loop nest structure and cache line lengths, to accurately reflect the number of cache evictions of the first-accessed variable that could be caused in each level of cache by the second-accessed variable. The numbers that were added to the affinity arrays were worst case numbers, based on the worst-case assumption that the two global variables have maximum overlap in the cache. As we examine each line in the cache map, it is not known how many lines of overlap exist. The worst case overlap is equal to the number of second-level cache lines occupied by the smaller global variable. In the example from the previous chapter, if A occupies 10 lines and B occupies 3, then there can only be 3 lines of overlap at most. Therefore, each observed line of overlap contributes one-third of the worst-case affinity measure to the conflict metric being computed. The scaling factors `proportion`$_i$ and `proportion`$_j$ are merely the reciprocals of the number of cache lines occupied by the variables with indices *i* and *j*. In the example given, these proportions would be 1/10 for A and 1/3 for B. The maximum overlap is determined by the smaller variable, which has the larger proportion, i.e. a single cache line is a larger proportion of the total size of the smaller variable than of the larger variable. In the example of A and B, one third of the worst-case (*i, j*) affinity, and one-third of the worst-case (*j, i*) affinity, are added to the cumulative conflict metric for each cache line that has both A and B mapped to it. This more accurately reflects the cache conflict than an all-or-nothing computation, which would provide no feedback to the local refinement to reduce the overlap between variables with high affinity if it is unable to completely eliminate the overlap.

Given the cache map data structure and the function to accurately compute a cache conflict metric for each cache level, a local refinement algorithm can be designed that has

a global view of the impact of rearrangements within a single partition, because the conflict metric is computed over the entire cache map. There are two fundamental operations used to perform the refinement: the *swap* and the *rotation*.

The swap operation selects two variables within a partition and swaps their positions within the partition. Because the variables could be of different sizes, this often means that all variables within the partition have their position within the cache map affected. The conflict metric is recomputed for the entire cache map, and the swap is undone if it did not reduce the conflict metric. For partitions with a small number of global variables (from 2 to 7), the swap is selected deterministically, as there are a small finite number of possible swaps to make within a small number of globals. (There are $C(n, 2) = n(n-1)/2$ choices of globals to swap in an n-global partition. For n=7, that is still only 21 choices for a swap from any given configuration of globals within the partition.) By keeping track of the count of bad swaps that have been tried and then undone since the last good swap, a simple switch statement can select among the (at most 21) choices for the next swap without repetition. For larger partitions, a uniform pseudorandom number generator is used to select a pair of variables to swap. In the deterministic case, the bad swap count tells us whether there are no more possible productive swaps because we have tried all possible swaps from the present ordering. For the larger partitions, a limit is set on the number of consecutive bad swaps (currently the limit is 60) to limit the time spent while still providing an opportunity for improvement.

When swapping terminates because no further progress is possible, the dilemma of the local optimum that is not a global optimum is faced. As with many heuristic search optimizations, it is possible to get into a region of the possible *solution space* (a.k.a. *search space*) of a problem in which all neighboring points in the solution space are less optimal than the present point, but there is a more distant point that is more optimal. There are two general approaches to breaking loose from the local optimum. The first is to permit movement through less optimal points in the solution space in order to have a chance to reach more distant points that are more optimal. The best current solution is recorded in memory so that it can be restored if the search fails to find a more optimal point. Simulated annealing is an example of a method that probabilistically permits this kind of movement to less optimal points, with the probability of permitting such movement declining over time as it set-

tles into the final region that it will search in the solution space. The expense of this general approach depends on the expense of saving the best current solution. In the present research, this would mean copying the order of variables in the present partition, or copying a portion of the cache map, and then restoring both the cache map and the order within the linked list that holds the global symbol table information. This is feasible but troublesome.

The second general approach is to permit the heuristic search to move from the current optimum to a distant point, rather than only permitting movement to neighboring points. This forces exploration of more of the solution space than would otherwise be permitted. Examples using this technique are *iterated hill climbing* and *tabu search* [Michalewicz and Fogel 2000]. The rotation operation was chosen for the current research because it is distinct enough from swapping to permit the exploration of more distant points in the solution space. For example, look at the possible configurations of a partition with three global variables, as shown in Figure 4.4 below. The global variables are numbered with 0, 1, and 2, and the six possible permutations of them are arranged so that 012 is considered the current configuration. The three configurations reachable in a single swap are connected by edges to the 012 node. The two configurations that are not reachable in a single swap are shown in the bottom row of the graph with no edges.

If the current configuration is a local optimum, and the three children nodes represent inferior configurations (i.e. they produce a higher conflict metric), then swapping will terminate after all three possible swaps have been tried and rejected. If the global optimum for this partition is found in one of the two bottom nodes, they can only be reached by permitting movement to a suboptimal node first, or by making a jump in the search space beyond the points reachable by a single swap. Each of the two can be reached from any of the three suboptimal child nodes in a single swap, but each can also be reached in a single rotation from the current configuration of 012. Rotating left by 1 position and by 2 positions gives exactly these two nodes. In this research, the rotations are tried in order and the configuration is then restored by a final rotation to the best rotated configuration that was found. This helps the local refinement process converge somewhat faster than accepting the first rotation that offers any improvement at all.

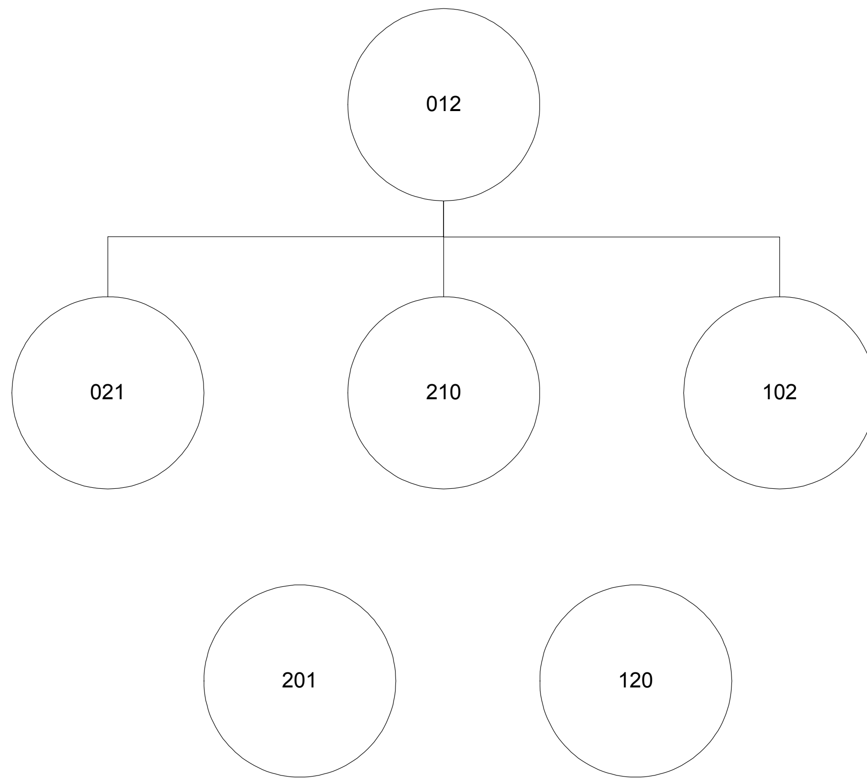


Figure 4.4
Solution space for 3-variable partition

Of course, the case for only three global variables in a partition is simpler, and the search space smaller, than for larger partitions. To see whether rotations provide any benefit in escaping local optima for larger partitions, first examine the next larger case, in which a partition has four variables. To avoid the messiness of a graph, the search space will be presented as lists of numbers:

0123	**0132**	***0312***	**1230**
	0213	***0231***	***1302***
	0321	***1032***	***2310***
	1023	***1203***	***2031***
	2103	***1320***	***3201***
	3120	***2013***	**3012**
		3021	
		3102	
		2130	
		<u>***3210***</u>	
		2301	

The first column contains a single configuration, 0123, which is the current configuration. The second column contains the six configurations reachable in a single swap from the current configuration. The third column contains the eleven new configurations reachable in a single swap from the second column; note that each element in the second column can also reach the first column by undoing the swap that led to it from the first column. The fourth column represents the six new configurations reachable in a single swap from the third column; again, all elements in the second column are reachable in a single swap from the third column. The configurations in bold font are the three configurations reachable by a rotation from the current configuration of 0123. Unlike the simpler case for a three-variable partition, not all configurations are reachable from the current configuration with one swap or one rotation. However, we are now stuck in a local optimum only if all configurations reachable with a rotation are inferior, as well as all that are reachable with a swap. If any rotation is more optimal, then swapping resumes in that new configuration. The configurations in italics are reachable in one swap from one of the three rotations of the current configuration. They constitute all configurations not already reachable in a single swap or rotation. The lone italicized and underlined configuration is reachable in a single swap from both of the rotations in the third column.

As the partition size grows, the number of sequential swaps and rotations required to reach every point in the search space from a given configuration grows. If the task were entirely to search a single permutation space, then swaps and rotations without permitting suboptimal intermediate points to be traversed would not be sufficient to avoid local optima. However, after the search space for a single partition has been searched to completion, the process moves on to the next partition, which will often have a different number of globals of various sizes and a different structure to its search space. Because the conflict metric is defined by interactions among partitions, the lack of further progress in one partition is observed in practice to be followed by significant progress in a later partition. Stated another way, if it is important to rearrange global number 3, global number 4, and global number 9 in a partition because of their conflict with a dozen other globals scattered throughout various other partitions, it is possible to move piecemeal towards that goal even if the local refinement of the present partition is incomplete when it has to terminate. As future work, the use of a more complex method such as a tabu search could be measured

against the present method of local refinement. It will be seen in the measurements in the next chapter that local refinement often contributes more to the total reduction of the conflict metric than the hierarchical graph partitioning process, which increases the confidence level in the design of the local refinement.

4.5 Padding Large Arrays to Minimize Conflicts

It was noted previously that global variables are classified into three groups by their size: smaller than the first-level cache, at least as big as the second-level cache, or in between. The largest arrays cannot be used in the graph partitioning process, as they each exceed the size of the largest partition. Some of these arrays can participate in the local refinement process by being placed in the cache map. For example, in Figure 4.1 at the beginning of this chapter, global array A wrapped around the cache because it was 1.5 times the cache size. If this cache were the second-level cache rather than the first-level cache as originally discussed, then A would not be used in the graph partitioning process. However, the argument made about Figure 4.1 still holds; it matters where other variables are placed relative to A, because not all of them have the same conflict with A. In the example, if B had more conflict with A than C or D had with A, then the placement in Figure 4.1(b) was judged superior to the placement in Figure 4.1(a), because it conflicted with the middle third of A rather than the first and last thirds of A.

In many programs, large arrays are processed by several different loops. In some cases, the loop iterates over the entire array. In other cases, the loop iterates until some condition is reached and the loop exits before processing the entire array. Intuitively, it is often the case that the early regions of the array are accessed more than the later regions over the course of the whole program's execution. Because it is always the case that an array that wraps around the cache at least once, but is not an exact multiple of the cache size, has its earliest region overlap more of its own elements than some other regions (witness the case of array A in Figure 4.1), then minimizing the conflict metric between a large array and another variable is accomplished by minimizing the overlap between the beginning of the large array and that other variable (as with variable B in Figure 4.1). This can be even more valuable than the conflict metric might indicate, as the conflict metric assumes that accesses are spread evenly across each global variable, while in practice the beginning of an array

can be accessed more intensively.

As a result, it is desirable to incorporate some of the largest arrays into the cache map. There are diminishing returns to the relative placement as the size of the array grows. In Figure 4.1, the conflict between B and A could be cut in half by better placement. If A were 2.5 times the cache size rather than 1.5 times the cache size, the reduction would be one third rather than one half, and the reductions would decrease asymptotically towards zero as A grows in size. Because there is a cost involved in increasing the size of the cache map, particularly in the computation of the conflict metric, which is performed after every swap and every rotation, along with the diminishing returns noted above, the current limit is to include arrays in the cache map only if they are three times the largest cache size or less. These arrays are not swapped or rotated, but can influence other swaps and rotations in other partitions. They are given a “partition” number higher than any of the other partitions, which means that all such arrays are sorted to the end of the globals list.

There is one thing that can be done about such large arrays. To keep any such array from mapping to the same starting cache line address as another array, creating the potential for thrashing if they are used in the same loop nest, padding equal to one second-level cache line is added at the end of one array to push the starting address of the next array out of direct conflict with any other large array. Using a pad size of more than a single cache line will be part of the experimentation in the next chapter.

4.6 Data Placement for Local Variables

Data placement by graph partitioning to reduce cache conflicts within the stack frames (i.e. the set of local variables) of a function can be performed in a similar manner as for global variables. There are several differences:

1. Very few functions have a stack frame large enough to exceed the size of the first-level data cache, and no stack frames have been encountered in this research that would exceed the size of a second-level cache. There is much less opportunity to reduce conflicts within a stack frame than conflicts between global variables.
2. Static data analysis for local variables does not need CFGs, inlining of CFGs, matching of actual parameters to formal parameters, etc. All that is needed is an execution frequency estimate for each basic block, followed by path analysis within the function.
3. Addressing cost reduction based on immediate addressing range has several differences, as described below.

The addressing of locals within a stack frame typically uses a *frame pointer* register as the base register. Late in the code generation process, the code generator emits assembly language definitions of the offset from the frame pointer for each local variable that is accessed from memory. (Some locals could be unused, and many others could be allocated to registers and not accessed from memory.) If a local variable has an offset that is farther away from the frame pointer than can be reached with an immediate address offset, then a more expensive addressing sequence is required.

For example, in the SPARC architecture, a local array called “dummy” could have one of its elements loaded from the stack frame into a register by the single instruction:

```
ld [%fp+dummy+308],%o1
```

The name “dummy” in this code is simply a constant offset from the frame pointer that was emitted by the code generator at the beginning of this function’s assembly language code:

```
dummy=-800
```

Thus, the load instruction is the same as:

```
ld [%fp+-800+308],%o1
```

This is resolved by the assembler through constant expression evaluation so that only one constant offset remains when the machine code is emitted by the assembler, which will be the machine code corresponding to:

```
ld [%fp-492],%o1
```

Note that the frame pointer points above the stack frame, and negative offsets are used to reach down to the local variables in the stack frame:

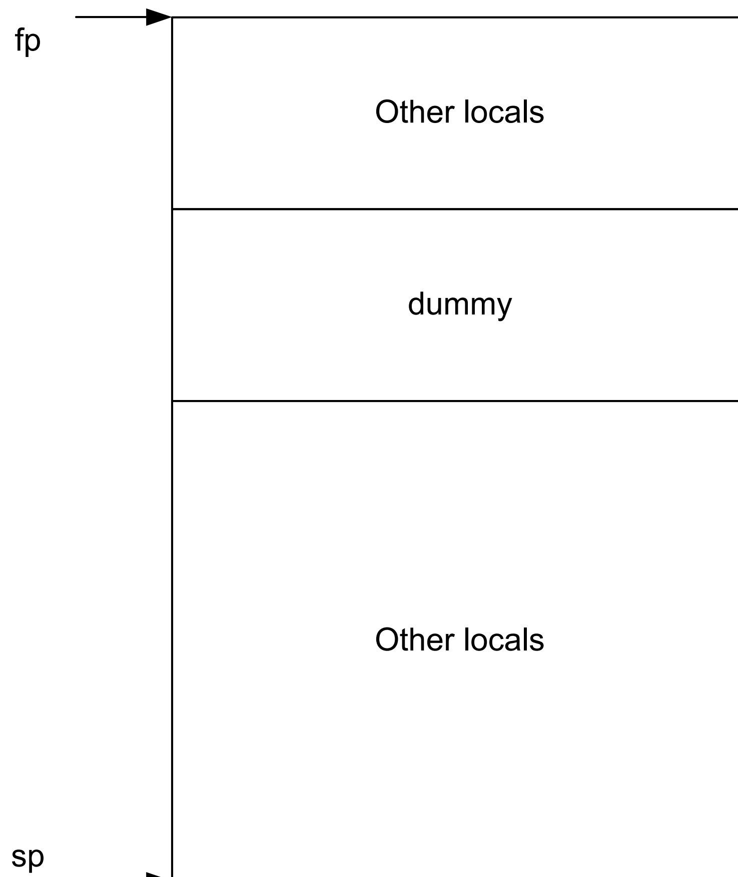


Figure 4.5
Simplified SPARC stack frame

The stack pointer (*sp* register in the figure) is used primarily to access certain regions at the bottom of the stack frame that include argument building areas, `alloca()` space (for quick dynamic allocation of small objects on the stack frame which are deallocated upon returning from the function), etc. The stack pointer might move in relation to the locals in the stack frame when `alloca()` calls are made, whereas the frame pointer (*fp* register in the figure) stays constant throughout the execution of the function. As a result, the frame pointer is used to access all local variables in most functions.

However, if a local variable gets beyond the immediate addressing range of the frame pointer, then an access become more expensive. In the example above, an offset 308 bytes into array “dummy” was loaded in a single instruction into register %o1. If “dummy” were farther away than 4KB from the frame pointer on a SPARC machine, the load would look like:

```
sethi %hi(dummy+308),%o7
add %o7,%lo(dummy+308),%o7
ld [%fp+%o7],%o1
```

This three-instruction sequence is required because only 13 bits are available for immediate addressing mode offsets in the SPARC instruction set, which gives an addressing range of plus or minus 4KB from a base register. With the frame pointer as the base register, only negative offsets can be used, meaning that any local variable whose starting address is more than 4KB below the frame pointer must be addressed using three instructions rather than one. The first instruction extracts the high-order 22 bits from the operand `(dummy+308)` and places them in a register that is only used in call and return sequences and is thus always available for short-term use within an expression evaluation. Because `sethi` only has two operands rather than the customary three, it is designed with a large immediate field to hold 22 bits. The second instruction uses the `%lo` operator to extract the lower-order 10 bits and add them to the temporary register, producing the full 32-bit address. The third instruction adds the temporary register to the frame pointer register to get the address of the data item, and loads it into a third register.

Two placement-related optimizations are motivated by this discussion. First, the local variables that are accessed the most would be good candidates for placing in the top 4KB of the stack frame, to minimize the dynamic frequency of the expensive addressing sequence. Second, the bottom 4KB of the stack frame could be addressed cheaply by the stack pointer, but only if the body of the procedure can be analyzed to ensure that no calls to `alloca()` are made, and no leaf-procedure optimization has changed the normal definition of registers such as the stack pointer or frame pointer. It would be desirable to move a second set of the most frequently accessed local variables down to the bottom of the stack frame, in those cases where the stack frame is large enough to not fall entirely within the reach of the stack pointer and frame pointer together; e.g., on the SPARC, when the stack frame exceeds 8KB, then priorities must be set for placing local variables within the upper and lower 4KB regions of the stack frame.

When the size of the stack frame exceeds the size of the first-level data cache (a rare occurrence), there will also be the need for data placement by graph partitioning and local refinement. In such a case, saving two instructions by moving a local variable to the bottom

of the stack frame and accessing it from the stack pointer could be more than negated by the costs of incurring more cache misses, if the bottom of the stack frame conflicts with the top of the stack frame or another heavily used local array somewhere in the stack frame. It will be shown in Chapter 6 that a cache miss typically costs more than two CPU cycles. As a result, cache conflict reduction shall be given priority over maximizing use of the stack pointer for cheap addressing whenever the two goals come into conflict.

The multi-goal optimization process can be described best by enumerating the different cases for stack frame sizes, using the SPARC as our example:

1. For stack frames of 4KB or less, all local variables are addressed cheaply already, and they do not conflict in the 16KB cache. No placement is needed.
2. For stack frames of 8KB or less, but more than 4KB, cheap addressing can be performed using a combination of the frame pointer and the stack pointer, and there are no cache conflict issues. The RTLs for variable accesses to some variables will need to be changed to use the stack pointer, and the constant emitted for the variable name will need to be changed from a negative offset, relative to the frame pointer, into a positive offset, relative to the stack pointer.
3. For stack frames greater than 8KB but less than or equal to 16KB, there are no cache conflict issues, but not all variables can be accessed cheaply, unless a certain stack pointer trick can be used which will be described below. The top 4KB needs to be treated as a knapsack to be packed with local variables. The benefit of each variable is its frequency of access, and the cost is its size. The bottom 4KB then needs to be treated as a second knapsack to be packed with the same criteria.
4. For stack frames greater than 16KB in size, placement by graph partitioning and local refinement needs to be done. The most heavily used 16KB partition needs to be placed closest to the frame pointer, and the top 4KB of that partition needs to be treated as a knapsack to be packed with local variables, as above. Whichever variables fall into the bottom 4KB need to have their addresses reassigned to use cheaper addressing based on the stack pointer.

The knapsack packing has been implemented with a greedy algorithm and also with an optimal dynamic programming algorithm. The space problems that sometimes arise with dynamic programming algorithms are avoided because the knapsack is a known small size (4KB) which is treated in units of first-level data cache lines of 32 bytes, making the knapsack capacity equal to only 256, the number of lines in the first-level cache. For such a small capacity, it is possible to solve the knapsack problem optimally and compare the solution time and solution quality to the greedy method.

The stack pointer addressing trick mentioned in item 3 above is as follows. It is ordinarily the case that cheap immediate-mode addressing can only be performed on a local variable whose entire address range lies within range of the base register used. For example,

if a 3KB array had a starting address slightly more than 4KB below the top of the stack frame, then its starting offset would need to be addressed expensively but most of its offsets could be addressed cheaply. To try to accomplish this would require extensive analysis of all RTLs in which the variable appears to determine which ones are adding a large enough immediate constant to come back within range of the frame pointer, e.g.:

```
dummy=-4300
ld [%fp+dummy+308],%o7
```

These constants would be folded by the assembler into a single constant value of -3992, which is within the -4096 range permitted for the single-instruction load shown. Because of the dangers of emitting illegal code if an analysis were to fail, this is not attempted. All RTLs that access a given local variable either use the cheaper addressing or the safer but longer sequence, based on the starting address of the variable.

This research makes an exception to this rule. There are many arrays that are accessed only by their base address added to a register offset. As a loop iterates over the array, the offset within the register is incremented in the loop, and no constant offsets are used for that variable, other than the constant defined for the name of the variable. This fact would not be helpful in the standard, prescribed addressing for local variables from the SPARC frame pointer, as the starting address would still be used as shown above, and if it fell out of the range of the frame pointer, it would need a three instruction addressing sequence before the index offset register could even be added to the address that was formed. Relative to the stack pointer, things are different. The variables in the stack frame grow up from the base address, which is closer to the stack pointer, to their end, which is closer to the frame pointer (refer again to Figure 4.5). If no constant offset is ever added to the name of the array, then the array can be placed near the top of the 4KB that falls within the immediate-mode addressing range from the stack pointer, such that most of the array extends beyond that 4KB limit. If careful analysis has proven that all offsets within the array are held in registers, not used as constants, then the array may be placed such that most of it lies outside the 4KB zone above the stack pointer.

This can have a positive effect for stack frames that are greater than 8KB. It might seem that an 11KB stack frame cannot entirely be addressed cheaply from the frame and stack pointers, as there will be a 3KB region that falls in the middle and is out of range of

the immediate addressing mode from either register. However, that space might be entirely filled by an array that uses the trick described. Or, the space might at least be partially filled by such a trick, thereby reducing the occurrences of expensive addressing.

One constraint upon our addressing optimization is the presence of unclear addressing expressions. For example, given two arrays called “sums” and “roots”, an RTL such as the following might be encountered:

```
ld [%fp+sums-roots],%o2
```

This can be created by a loop-based optimization, such as induction variable elimination, which is attempting to use a single offset register to index into both arrays within a loop. The optimization assumed that these two names would be emitted as offsets relative to the frame pointer, and the above RTL will only work if that is the case, because the RTL is trying to use those offsets to compute the distance between the two arrays. Note that it is permissible for data placement to move these arrays; they do not have to be adjacent or keep a fixed distance. But the distance between them needs to be computable using their named offsets. If the addressing mode optimization were to move “roots” down to the bottom and start addressing it using the stack pointer, while “sums” continues to be addressed using the frame pointer, then the sequence of statements that reference these arrays will look like:

```
sums=-788
roots=3424
ld [%fp+sums-roots],%o2
```

These constants are nothing like what was envisioned by the induction variable elimination optimization, and the code produced will have an offset that is out of immediate offset range and will be illegal (in addition to the fact that the offset would be a nonsensical combination of *fp*-based offset and *sp*-based offset, even if it were legal). To avoid this problem, any local variable that is seen as part of any addressing expression in which a register value is subtracted from it, or another variable’s address is added to or subtracted from it, is tagged in the local symbol list and will be addressed from the frame pointer, regardless of where it is placed. If the stack frame is not big enough to exceed the cache size, then it will be placed near the top of the stack frame.

4.7 Reducing Bus Cycles with Load Coalescing

The benefit of reducing bus cycles by coalescing loads or stores is machine dependent. In order to measure the potential benefit, microbenchmarks were written and timed for the UltraSparc-IIi systems. These microbenchmarks are described at the end of Chapter 6. The results can be summarized briefly. Coalescing 32-bit integer loads into 64-bit integer loads does not reduce execution time. The combination of a 64-bit load and a 64-bit shift instruction to unpack the results of the load must consume as much time as the two 32-bit loads that they replaced. Coalescing integer stores increases execution time, because two instructions are required to pack the 32-bit values into a 64-bit register before the store. Coalescing 32-bit floating-point loads into a single 64-bit floating-point load does not reduce execution time, contrary to explicit statements in the vendor documentation, and despite the fact that no unpacking is required in the floating-point registers. Coalescing 32-bit stores into a single 64-bit store does reduce execution time. It is questionable whether significant time savings can be achieved by implementing this coalescing optimization alone, but this will be a subject of ongoing study. it is possible that coalescing load or store buffers are reducing the benefits of these optimizations. Continued development of microbenchmarks might reveal more about which coalescing optimizations are beneficial in which circumstances.

On architectures which benefit from load or store coalescing, the improvement is greatly increased by combining the coalescing with loop unrolling, as in the prior work of Davidson and Jinturkar [Davidson and Jinturkar 1994]. This work would be enhanced by the data placement optimizations in this research, because the alignment of all arrays would be known at compile time, having been fixed by the data placement optimization. As a result, the alignment concerns that had to be addressed before each application of the unrolling and coalescing optimization would be greatly simplified. There is also potential for simplification through the interprocedural operations of the CFG inlining, which could provide feedback from the matching of actual to formal parameters in order to resolve the question of whether two parameter arrays overlap in memory and create a data hazard.

4.8 Reducing Addressing Costs for Global Variables

The framework for reducing addressing costs for global variables has been laid through the increase in data locality accomplished by global data placement. Implementation of this optimization is ongoing work that will not be presented at this time.

4.9 Reducing Code/Data Conflicts in Unified Caches

It will be seen in the next chapter that programs that have been optimized through data placement will receive increased or decreased conflicts between code and data in the unified second-level cache. The increases or decreases balance out, but there is potential to achieve better than neutral performance in this regard by placing code so that it has less conflict with data after the completion of data placement.

The design of this optimization is facilitated by the static data affinity analysis, which prepared for this optimization by recording the function name in each CFG node and then preserving that name throughout the inlining process. At the conclusion of the CFG inlining, the final whole-program CFG can therefore be used to build a code-versus-data temporal affinity array through a modified form of the path analysis step. This new affinity array would be passed down to the *vpo* pass through the program, which would record the order and size of each function during final code generation.

A simple way to use this information would be to measure the conflict of all N possible starting locations for the code in the virtual memory address space, where N is the number of cache lines in the second level cache. A linker script could then be output that would fix the starting addresses of the code and data to minimize their conflict. This approach involves no rearranging of code, hence it does not target the first-level instruction cache misses that are usually targeted by code placement optimizations.

A more comprehensive approach would be to perform a complete code placement optimization, then perform the starting address determination afterwards. This would require analysis at a finer granularity than the static data affinity analysis currently offers if it were to compete with prior code placement efforts. The best such efforts have used dynamic profiling runs of a very long duration, as discussed in Chapter 2. Because one of the choices in this research has been to make the optimizations practical and usable without

profiling runs that would deter most users from invoking the optimizations, the follow-up work to this dissertation will include the simpler approach to code placement in order to measure how much benefit can be obtained in the second-level cache alone.

4.10 Code Generation after Data Placement

In *vpo* code generation without data placement, the assembly language code for each function is generated from the optimized RTLs at the end of the final optimizations. The generated assembly language code includes assembly language that was generated by the front end of the *lcc* compiler and carried along within the RTLs as special assembly language lines that are not subject to the RTL optimizations. These special assembly language lines include the declaration and initialization lines for global variables.

When the data placement framework is selected by command line switches passed to *vpo*, the assembly language lines for global variables are removed during final code generation and attached to the global symbol table entries for the corresponding global variables. Instead of emitting these lines wherever they happen to fall in the program text, they are all emitted at the end of the assembly language file, with appropriate machine-dependent alignment directives to ensure that packed cache lines and arrays are aligned on second-level cache line boundaries. This completes the operation of the data placement framework for the application program.

Chapter 5

Measurements and Results

This chapter presents measurements of the results of the optimizations. First, a set of benchmark programs is presented, and the rationale for their selection in this research is given. The benchmark programs are measured to give an understanding of their relevant characteristics prior to application of the optimizations. Next, the results of applying the optimizations are measured using key criteria, such as execution time and cache misses. Then the operation and effectiveness of individual components of this research, such as the static data analysis and particular optimization stages, are measured and discussed in light of the results. Next, the results and measurements are compared with prior work. Finally, what the measurements suggest should be done to enhance this research in the future is discussed.

5.1 Benchmark Programs

The selection of benchmark programs follows from certain characteristics of the research. Because the static data analysis does not include the heap (i.e. dynamically allocated) variables, programs that make significant use of heap memory are poor candidates for measurement of this work. The conflicts among heap variables, and between heap and statically allocated variables, will be unmanaged and random in their effects. In some cases, a seemingly efficient reordering of the statically allocated variables to reduce their conflict among themselves will accidentally increase their conflict with heap variables. In other cases, the conflict with heap variables will be reduced accidentally, and the data placement optimization will receive undeserved credit for the resulting speedup.

This phenomenon was actually observed in a pathological case during early testing of the data placement optimization. Due to a coding error, the assembly language code generation stage was accidentally performing a rearrangement that was essentially random. The programs being used for testing included 14 programs from the SPEC CPU2000

benchmark suite ([Standard Performance Evaluation Corporation 2002]) that are written in the C language. Each of these programs makes significant use of heap memory, as is typical of programs written in the C language. Due to the coding error, most of these programs showed a decline in performance. The floating-point intensive program *179.art* showed a speed-up ratio of 1.37, which corresponds to a 27% reduction in execution time. Because the statically allocated data within this program has a total size that is far less than the first-level data cache capacity, conflicts among statically allocated variables could not be the reason for the speedup. Because rearrangements within the statically allocated data section do not change the starting address or size of the section, they will not affect the addresses assigned to heap variables, so a reduction in conflicts between heap variables could not be the reason for the speedup. Only conflicts between statically and dynamically allocated variables would seem to be a plausible source of the huge difference in execution times. Very small changes in the starting address of the static data region, accomplished experimentally by adding some padding bytes, caused the speedup to disappear. Future work might be able to isolate which static and dynamic variables are involved in a cache thrashing conflict.

Because of the significant use of heap variables in the SPEC benchmarks written in C, they were not considered useful for this work other than as test cases. There are also other problems with these codes, such as indirect function calls through function pointers (which creates a hole in the static profile, as *vpo* does not have the kind of analyses needed to resolve the target of these calls at compile time), and a few cases in which the control flow graph is irreducible (i.e. *spaghetti code*, in which `goto` statements prevent a clean analysis and assignment of execution frequency estimates). In the final test run of these fourteen programs, eight of them showed modest changes in the benchmark score. With all scores on the 500 MHz UltraSparc-IIi machines ranging from 100 to 170, the eight score changes were -1 (once), +1 (twice), +2 (three times), and +3 (twice). While it is comforting to have a positive effect overall, there is too much about these benchmarks that the present analysis and placement work is not able to scientifically manage, precluding the drawing of any conclusions from this set of programs.

While testing and developing the optimizations, C language applications that did not suffer from these problems were sought. Few C codes fit this description, however. The

older data compression application *compact* was found to be suitable, and it will be used in the measurements in this chapter. A number of small synthetic benchmarks were used in testing the correctness of the placement optimizations, along with a few kernels. However, data placement only makes sense within the context of a whole application program, not a kernel or microbenchmark.

Given the requirements of our static data analysis, Fortran benchmarks are natural choices. The important characteristics of the typical Fortran code with respect to data placement are:

- No heap variables (except in very recent versions of Fortran).
- Even local variables are statically allocated, rather than stack allocated (except in recent Fortran extensions that allow recursion), which increases the pool of data available to the global data placement optimization.
- No calls through function pointers to create inaccuracy in the analysis and inlining process.

By avoiding the most recent Fortran variants that permit extensions that are reminiscent of the C language, it is possible to obtain a collection of suitable benchmarks for this research. One potential disadvantage in such older Fortran programs is the presence of irreducible control flow graphs due to extensive use of `goto` statements.

Because *vpo* has only C language front ends, the use of the *f2c* Fortran to C translator is necessary. The *f2c* translator is limited to the Fortran 77 language standard, which includes previous standard dialects, with a few extensions that enable *f2c* to translate the extensions to Fortran 77 that were common in the compilers from various major vendors. One quirk of *f2c* is that it takes all of the variables within a Fortran COMMON block and wraps them up into a C `struct`, meaning that it has turned a list of variables into a single variable. This apparently simplifies the workings of *f2c* in some way, but it is highly undesirable for a data placement optimization, which needs the flexibility of rearranging these variables anywhere in memory that is most efficient. To deal with this, the `structs` are unbundled within a text editor using simple search and replace commands after *f2c* has produced the C code.

A key point in the decision to use *f2c* was the treatment of the data access patterns of the Fortran program during translation by *f2c*. Fortran programs have a *column-major* array layout, whereas C programs have a *row-major* array layout. This means that a two-dimensional array in the C language with three rows and three columns would have the following memory addresses assignment order for its elements: (row 0, column 0); (row 0, column 1); (row 0, column 2); (row 1, column 0); (row 1, column 1); etc. In other words, to traverse the array elements in the order in which they are laid out in memory, a program would traverse the first row, then the second row, etc. By contrast, the address assignment order in Fortran (which starts numbering rows and columns at one instead of zero) would be: (row 1, column 1); (row 2, column1); (row 3, column 1); (row 1, column 2); etc. To traverse a Fortran array in memory assignment order, a program would iterate down the first column, then the second column, etc.

It would be highly undesirable for a Fortran program that was written to carefully traverse its arrays in column order to be translated to a C program that also traversed the arrays in column order, because column-order traversal in C would not use consecutive memory addresses. To avoid changing the access order of arrays, *f2c* transforms all Fortran arrays into a single dimension and generates the subscript arithmetic to access the elements in the same memory address order as was used by the original Fortran program. This ensures that the opportunities for data placement improvements are neither helped nor harmed by operating on the translated form of the program rather than the original Fortran version.

The Perfect Club benchmark suite ([Cybenko et al. 1990]) is a set of thirteen Fortran 77 codes developed originally to test the performance of supercomputers. After several years of development, the effort was merged into the new SPEC benchmarking consortium, which used modified versions of some of the Perfect Club programs in its collection of floating-point Fortran benchmarks. One item left unfinished in the Perfect Club effort was the development of a second set of input data sets that would have longer run times on supercomputers. Because it was initially considered desirable to be able to run the Perfect Club programs in a reasonable time on non-supercomputer benchmark systems that were commonly used for older benchmarking efforts, such as the VAX 11/780, the input data sets were small. The implication is that the code space of the programs gets exercised by the

input data set, but the data space might not be exercised as thoroughly as the code space. Even so, it should be possible for an effective data placement optimization to demonstrate reductions in cache misses and execution time, though perhaps to a lesser degree than would be possible with larger input data sets.

Table 5-1 lists the benchmarks used for measurements in the remainder of this chapter. Listed are the compact program, mentioned earlier, and ten of the thirteen Perfect Club benchmarks. One of the Perfect Club benchmarks (*OCEAN*) uses non-standard extensions to the Fortran 77 language and causes an error within *f2c* during translation. The other two Perfect Club benchmarks not shown (*SPICE* and *QCD*) have irreducible control flow graphs that make a thorough analysis and data affinity analysis impossible. It is possible that these problems could be overcome in the future through a manual recoding of the source code for these benchmarks. This has not yet been done, so these benchmarks will not be mentioned any further.

Benchmark	Description	Code Lines
compact	**Data compression**	**493**
ADM	**Air pollution model**	**6105**
ARC2D	**2-D fluid flow**	**3964**
BDNA	**Molecular dynamics of DNA**	**3977**
DYFESM	**Structural dynamics**	**7608**
FLO52	**Transonic flow past airfoil**	**1986**
MDG	**Molecular dynamics of water**	**1238**
MG3D	**Seismic depth migration**	**2812**
SPEC77	**Weather simulation**	**3885**
TRACK	**Missile tracking**	**3784**
TRFD	**Electron orbital transforms**	**485**

TABLE 5-1: Benchmark programs.

The number of lines of code is of much less relevance to data placement optimizations than the data characteristics of the benchmark programs. Table 5-2 divides the data space of the benchmark programs into three categories. The first category is composed of all variables each having a size less that the first-level cache size. These variables provide

Benchmark	size < L1	L1 <= size < L2	L2 <= size
compact	**18,496** 100.0%	**0** 0.0%	**0** 0.0%
ADM	**23,040** 3.70%	**0** 0.0%	**600,000** 96.30%
ARC2D	**20,764** 0.26%	**3,505,920** 43.53%	**4,527,040** 56.21%
BDNA	**716,096** 33.92%	**160,000** 7.58%	**1,234,944** 58.50%
DYFESM	**129,088** 100%	**0** 0.0%	**0** 0.0%
FLO52	**44,096** 4.19%	**1,007,936** 95.81%	**0** 0.0%
MDG	**18,496** 7.68%	**222,272** 92.32%	**0** 0.0%
MG3D	**34,624** 1.80%	**1,408,128** 73.19%	**481,280** 25.01%
SPEC77	**199,360** 15.59%	**1,079,680** 84.41%	**0** 0.0%
TRACK	**195,904** 28.45%	**492,800** 71.55%	**0** 0.0%
TRFD	**3,456** 0.04%	**0** 0.0%	**8,000,000** 99.96%

TABLE 5-2: Cumulative sizes of global variables by category.

the greatest opportunity for effective application of data placement optimizations, as they are subject to all partitioning stages, the local refinement process, and (for some) being packed into cache lines. The second category is composed of all variables less than the second-level cache size, but greater than or equal to the first-level cache size. These variables participate in the first partitioning stage and the local refinement process, but not the finer-grained partitioning, nor the packing of scalars into cache lines. The third category is composed of all variables each of which is at least as big as the second-level cache. No matter where these variables are placed, each conflicts with all other variables in the system in all cache levels. In the absence of array subscript analyses, such as those developed by Pugh and others [Pugh 1994], these variables contribute cache conflict that is unmanageable by

the present research. Array subscript analysis will be discussed as future work later in this dissertation.

As discussed, the higher the percentage of global data that falls in the first category, the better for the data placement optimizations. The higher the percentage that falls in the third category, the worse for the data placement optimizations. It can be seen that three of the benchmark programs (ADM, ARC2D, and TRFD) have the unfortunate combination of a high percentage in the third category and a low percentage in the first category. Another key factor in enabling effective data placement optimizations is a high level of cache conflict in the global data layout before data placement. These factors will be discussed in more detail below.

5.2 Performance Results for Optimizations

In this section, the performance results of the data placement optimizations for the chosen benchmark programs will be presented from a variety of measurements. First, the execution time speedup will be shown. Then, the reductions in first-level and second-level cache misses will be presented, based on cache simulations over complete executions of the benchmark programs. Further cache simulations will analyze the effects of conflicts between code and data in the unified second-level cache. In order to understand the relationship between cache miss reductions and execution time speedups, the cache miss reductions will be presented again, this time in terms of misses per instruction executed, and the before and after first-level cache miss rates are presented for the same purpose. A comparison between the effects of data placement and the effects of increasing the associativity of the data cache is given in order to understand the significance of the improvement made by software (compiler optimizations) as compared to hardware improvements that could be quite expensive. Finally, the benefits of the data placement framework are shown for two aspects of performance that were only indirectly addressed by the design of the framework, namely first-level data cache write misses and data TLB misses.

Table 5-3 shows the execution time speedup for the benchmark programs after applying hierarchical data placement, without bus or address range optimizations. Discus-

Program	Time before placement	Time after placement	Speedup
compact	21.52	19.65	1.095
ADM	11.49	11.25	1.021
ARC2D	98.39	97.06	1.014
BDNA	17.97	16.79	1.070
DYFESM	10.97	10.71	1.024
FLO52	18.33	17.78	1.031
MDG	62.35	61.22	1.018
MG3D	326.45	315.76	1.034
SPEC77	65.94	64.75	1.018
TRACK	2.30	2.25	1.022
TRFD	14.59	14.43	1.011

TABLE 5-3: Execution time speedup results.

sion of these additional optimizations is deferred until later in this section. All times are in user seconds as measured by the best time out of five timings of the programs on an unloaded system, using the *time* command. The speedup is the ratio of the time before data placement to the time after data placement. Measurements in percentage of execution time reduced would be slightly lower, i.e. a speedup of 1.095 corresponds to a time reduction of 0.089, or 8.9%, a speedup of 1.034 corresponds to a time reduction of 3.3%, etc.

All measurements of *compact* are for a compression of an input file of size 27,042,320 bytes into an output file of size 9,957,412 bytes.

The shortest run time, by far, is for the Perfect Club benchmark program *TRACK*. The Unix *time* command uses a system clock with a granularity of one sixtieth of a second. A quantization error could therefore account for a full one-third of the execution time reduction in this benchmark. As a result, this benchmark should be considered too short in run time to contribute towards valid conclusions. It will be shown in the remaining measurements, but this drawback should be considered in evaluating its results.

In order to corroborate these timings, and especially to deepen our understanding of the effects of the data placement optimization, cache simulations using the *cachesim5* tool in the *Shade* toolkit [Cmelik and Keppel 1994] have been run on all programs. All tools in the *Shade* toolkit, including *cachesim5*, execute a given UltraSparc executable through

direct interpretation. Table 5-4 shows the first-level data cache miss reductions for the benchmark programs. Because the first-level data cache is a write-through, no-write-allo-

Program	L1D read misses before placement	L1D read misses after placement	Change in %
compact	32,624,810	1,987,451	-93.91
ADM	39,670,130	28,724,147	-27.59
ARC2D	606,974,940	583,434,147	-3.88
BDNA	173,319,668	108,755,314	-37.25
DYFESM	20,576,223	12,005,278	-41.65
FLO52	77,440,778	72,389,112	-6.52
MDG	148,673,528	89,405,296	-39.86
MG3D	1,794,685,088	1,323,789,402	-26.24
SPEC77	215,766,128	194,999,193	-9.62
TRACK	7,399,887	6,837,354	-7.60
TRFD	36,930,009	30,873,904	-16.40

TABLE 5-4: First-level data cache read misses.

cate cache, there is no attempt to reduce write misses to the first-level data cache. As discussed in Chapter 3, stores are not part of the path analysis and first-level cache affinity array for this reason. Examination of the cache simulation data confirms that reductions in first-level cache write misses were achieved for every program, in roughly the same proportion as the read miss reductions, but write miss reductions do not seem to provide reductions in execution time on microbenchmarks.

The very good results shown so far for the *compact* program warrant special examination. As shown in Table 5-2, *compact* only has a little more global variable space (18KB) than the capacity of the first-level data cache (16KB). This would suggest that there should only be a small opportunity for data cache conflict reduction. Instead, a speedup of 1.095 and a 93.91% reduction in first-level data cache read misses are achieved. Most of the global variable space of *compact* is occupied by two large buffers, one for input data and one for the compressed output data. The remaining global variables are scalars and small formatting strings for input and output operations. The buffers are accessed sequentially and

should benefit from spatial locality in the cache, minimizing the possibility of thrashing. In addition, as will be shown later, the entire cache conflict reduction is achieved through the cache line packing stage of the data placement optimization, after which there is no conflict in the cache map and hence no further stages are invoked.

The most plausible explanation for the speedup in *compact* is that there was cache line thrashing among scalars that was entirely eliminated by cache line packing and array alignment. Cache line packing reduces compulsory misses by increasing temporal locality within cache lines. This increase in locality can also have a positive effect on reducing capacity misses, as the reduction in cache lines that are only partly used can reduce the number of cache lines in the working set of the program at any point in time. This could be significant for a program whose global data space barely exceeds the capacity of the first level cache. Also, there can be a reduction in conflict misses by moving scalars with high temporal affinity into the same cache line, if they previously occupied conflicting lines. Finally, there are microarchitectural problems with a phenomenon known as *cache miss jamming* [Bacon et al. 1994]. Cache miss jamming is high temporal locality among cache misses; that is, if the cache misses of a program are not spread very evenly across the execution time of the program, then a burst of cache misses can occur in a short time, alternating with a long time with few misses. During the short burst of cache misses, Bacon et al. observed that certain microarchitectural features, such as branch prediction tables, branch target buffers, load buffers, write buffers, etc., can only handle a limited number of outstanding cache misses. A non-blocking cache, for example, can work in conjunction with load and write buffers to continue execution of the instruction stream while cache misses are awaiting service [Chen and Baer 1992]. This masks much of the effect of the cache misses. When internal limits are reached, the CPU must be stalled until a certain number of misses are serviced and removed from the list of outstanding misses.

Another possibility is that a load instruction can be followed immediately by a use of the loaded value. If the load creates a cache hit, there will only be a delay of one CPU cycle before the loaded value is available for use. If the load creates a cache miss in the first-level cache and a cache hit occurs in the second-level cache when the cache miss is serviced, then a 10-cycle stall will occur if the next instruction needs to use the loaded value (section 21.3.5 of [Sun Microsystems 1997]). As a result, the instruction scheduler optimi-

zation in the compiler will try to space loads and uses of loaded values as far apart as possible. In a tight loop, it is not possible to create much distance between the load and the use. A cache thrashing conflict in a tight loop could create an unusual number of stall cycles, and the period of little or no conflict that follows this burst of cache misses might make the cache simulation appear similar to another program that actually has far fewer stall cycles.

Thus, not all cache misses have the same effect on execution time. It seems likely that *compact* was suffering from an unusually bad pattern of cache misses (i.e. bursts) and uses of the values being supplied by the loads that caused the cache misses. The speedup of compact is excellent validation of the cache line packing stage of the data placement optimization, but it must be viewed as an unusually good result in this respect. The effectiveness of the cache line packing stage will be demonstrated later on a variety of programs.

Tables 5-5 through 5-7 show the effect on second-level cache read misses, write

Program	L2 data read misses before placement	L2 data read misses after placement	Change
compact	1,683	1,682	-0.06%
ADM	2,370,668	961,701	-59.43%
ARC2D	106,550,108	116,107,075	+8.97%
BDNA	11,726,880	9,318,571	-20.54%
DYFESM	545,988	716,876	+31.30%
FLO52	10,900,222	5,882,753	-46.03%
MDG	451,886	190,717	-57.80%
MG3D	289,621,453	294,325,340	+1.62%
SPEC77	24,306,315	21,613,586	-11.08%
TRACK	411,519	731,166	+77.67%
TRFD	664,333	651,288	-1.96%

TABLE 5-5: Second-level unified cache data read misses.

misses, and total misses, respectively. Whereas the first-level cache results were uniformly reductions in misses, the results vary for the second-level cache. This fact can be attributed to several factors:

Program	L2 data write misses before placement	L2 data write misses after placement	Change
compact	729	723	-0.82%
ADM	313,941	230,396	-26.61%
ARC2D	33,722,959	32,896,106	-2.45%
BDNA	4,823,524	2,164,759	-55.12%
DYFESM	25,734	17,200	-33.16%
FLO52	3,785,363	3,402,061	-10.13%
MDG	171,822	11,829	-93.12%
MG3D	78,696,804	162,795,396	+106.86%
SPEC77	3,545,927	3,082,617	-13.07%
TRACK	307,697	584,020	+89.80%
TRFD	233,127	225,943	-3.08%

TABLE 5-6: Second-level unified cache data write misses.

Program	L2 data total misses before placement	L2 data total misses after placement	Change
compact	2,412	2,405	-0.29%
ADM	2,684,609	1,192,097	-55.60%
ARC2D	140,273,067	149,003,181	+6.22%
BDNA	16,550,404	11,483,330	-30.62%
DYFESM	571,722	734,076	+28.40%
FLO52	14,685,585	9,284,814	-36.78%
MDG	623,708	202,546	-67.53%
MG3D	368,318,257	457,120,736	+24.11%
SPEC77	27,852,242	24,696,203	-11.33%
TRACK	719,216	1,315,186	+82.86%
TRFD	897,460	877,231	-2.25%

TABLE 5-7: Second-level unified cache total data misses.

1. Second-level cache misses are more than an order of magnitude less frequent than first-level cache misses, on average. Even with second-level cache miss penalties being weighted by a factor of eight in comparison to first-level misses, the key to performance improvement still lies more in reducing first-level cache conflict, and this is accurately reflected in the local refinement stage of data placement.

2. In the absence of array subscript analysis, there is imprecision in measuring the conflict between large arrays. If only the first 15% of an array is actually used in a test run of a benchmark program, then treating the access pattern recorded in the data affinity profile as uniform across the array is imprecise and can lead to conflicts. As 15% of a large array might still fill the first-level cache but not the second-level cache, this imprecision affects the second-level cache disproportionately.
3. The rather small test inputs for the Perfect Club benchmark suite do not necessarily exercise the data space in a way that corresponds well to the static data affinity analysis. This will be more of a problem on larger variables than on smaller ones, and hence will have a greater impact on the second-level cache.

The only solutions to these problems are to incorporate prior work in array subscript analysis into the data affinity analysis, and to find Fortran benchmarks that are more suitable and have larger test inputs. The former solution is a major compiler engineering project. The approach of this research has been to emphasize the creative development of original ideas, rather than to engineer the best possible compiler by implementing the published ideas of others. Integration of present research with past research is best left to a future enhancement effort. The latter solution is hampered by the fact that, lacking a Fortran front end for *vpo*, all Fortran-90 programs, and many recent Fortran-77 programs that use extensions to the published language standard for Fortran-77, are not able to be translated by *f2c*. This rules out almost all of the Fortran programs in the SPEC CPU2000 benchmark suite, which would otherwise be suitable because of the longer run times and larger input data sets than are found with the Perfect Club programs. Searching for additional and better benchmarks is part of an ongoing effort to enhance this research.

How can the range of results within the second-level cache be explained for each individual program? How do the general considerations mentioned above apply in each case? The only increases in second-level total data misses are in the four programs *ARC2D*, *DYFESM*, *MG3D*, and *TRACK*. The problem of a very small input data set, which is not likely to match the static data affinity profile, is very likely to apply to *TRACK*. This program has such a short run time that its timing results are of questionable value, as previously noted. It has more than five times the data of *DYFESM*, for example, yet has only one fifth the execution time. This program had about three times as much increase in second-level cache data misses as the next worst program. *MG3D* and *ARC2D* have tiny percentages of global data space in variables smaller than the first-level cache (1.80% and 0.26%, respectively, as seen in Table 5-2), making them difficult to optimize in the absence of array

subscript analysis of their large arrays. Even with these difficulties, the percentage increase in second-level data misses is small for *ARC2D*.

The most disappointing result is for *DYFESM*, which demands closer scrutiny. This program has 100% of its global data space in the first category in Table 5-2, making it an excellent candidate for data placement optimizations. Yet, this program yields only a little over 2% speedup, and seems to have lost part of its potential speedup (judging by its 41.65% reduction in first-level cache read misses) due to the increase in misses in the second-level cache.

Notice that *DYFESM* has a total data space of only 129,088 bytes, while the UltraSparc-IIi has a second-level cache size of 256KB. Therefore, the increase in second-level cache data misses can only have occurred because of an increase in conflicts between code and data lines in the second-level cache, which is a unified cache. This is an indication that there is a need to enhance this research by integrating a code placement implementation with the existing data placement optimizations. The foundation for such an integration was laid in the static data affinity analysis, which recorded the function name in which each data access list was originally found, as described in Chapter 3. As a result, after inlining the CFGs of all functions back up to the `main()` (or other entry point) function, a total view of which functions access which data items can be obtained. Thoughts on using this profile to reduce conflicts of code and data in the second-level cache are given at the end of this chapter.

Confirmation of the hypothesis that code and data lines are colliding in the unified second level cache at a higher rate after data placement can be obtained from cache simulations. One result of effective data placement should be to smooth out the conflict across the cache lines, i.e. reduce the range of values of the conflict metric computed for each cache line. Rather than having very high conflict in a few cache "hot spots" and little or no cache conflict elsewhere, the variance in conflict metric values across individual cache lines in the cache map should be reduced as a side effect of reducing the sum of those values. What effect will this have on conflicts between code and data in the unified second-level cache? The answer depends on the mapping of code to second-level cache lines. Because this is not a mapping that is being managed by the data placement optimizations, the effect should be essentially random and average out to zero over a large set of programs,

but the effect on a particular program could range from a significant reduction in code versus data conflicts to a significant increase in such conflicts.

Table 5-8 shows the changes in instruction cache misses in each level of cache, along with the change in *back invalidation* of first-level instruction cache lines. Because

Program	Change in L1 instruction cache misses	Change in L2 instruction cache misses	Change in L1 instruction back invalidations
compact	-0.46%	+0.33%	-0.45%
ADM	-27.26%	-55.04%	-61.69%
ARC2D	+6.23%	+6.06%	+6.96%
BDNA	+6.68%	+23.13%	+29.32%
DYFESM	+31.93%	+26.12%	+44.07%
FLO52	+1.39%	+3.64%	+1.66%
MDG	+5.77%	+6.16%	+9.43%
MG3D	-13.06%	-9.21%	-14.46%
SPEC77	-10.95%	-31.61%	-44.88%
TRACK	+13.11%	+161.18%	+262.39%
TRFD	-1.40%	-1.39%	-2.19%

TABLE 5-8: Changes in instruction cache misses and invalidations.

the second-level cache has the inclusion property for both code and data, i.e. all contents of the two first-level caches are guaranteed to be included in the second-level cache as well, it is sometimes necessary to evict one or more cache lines from a first-level cache. For example, if a 64-byte instruction line is being evicted from the second-level unified cache, then the two corresponding 32-byte lines in the first-level instruction cache (if they are present; inclusiveness is an asymmetric relation) must be evicted to maintain the inclusion property.

It is apparent from Table 5-8 that there is a wide range of accidental costs and benefits from conflicts between code and data in the unified second-level cache. It is noteworthy that *BDNA*, which achieved the highest speedup among the Perfect Club benchmark programs, did so despite suffering an increase in conflicts between code and data in the sec-

ond-level cache, which increased back invalidations in the first-level instruction cache, which in turn increased misses in the first-level instruction cache.

In order to judge the significance of the percent changed data that have been presented in the last five tables, and relate these change percentages to the speedup results presented in Table 5-3, the numbers must be calculated as a percentage of instructions executed. If a first-level data cache read miss happens every 50 instructions executed, and data placement cuts that frequency in half, then the reduction is more significant than cutting first-level data cache read misses in half for a program that only has one such miss per 300 instructions before data placement. Some of the larger percentage numbers that have been shown are either more encouraging or more alarming than they are when calculated as a percentage of instructions executed.

Table 5-8 shows the changes in first-level data cache read misses, second-level cache total data misses, and data cache back invalidations (not previously shown in any table above) as percentages of instructions executed. Values that round to 0.000% are given

Program	Change in L1 data cache read misses	Change in L2 data cache total misses	Change in L1 data back invalidations
compact	-0.315%	-0.000%	+0.000%
ADM	-0.312%	-0.043%	-0.081%
ARC2D	-0.130%	+0.048%	+0.038%
BDNA	-1.022%	-0.080%	-0.066%
DYFESM	-0.272%	+0.005%	+0.057%
FLO52	-0.071%	-0.076%	-0.009%
MDG	-0.269%	-0.002%	-0.002%
MG3D	-0.424%	+0.080%	-0.016%
SPEC77	-0.111%	-0.017%	-0.023%
TRACK	-0.078%	+0.083%	+0.148%
TRFD	-0.127%	-0.000%	-0.000%

TABLE 5-9: Data cache changes as a percentage of executed instructions.

a plus or minus sign to indicate whether the very small change was an increase or decrease.

A comparison of Table 5-9 with the speedup values in Table 5-3 shows a correlation between the two sets of numbers. The Perfect Club program with the highest speedup (by a wide margin), *BDNA*, had the largest reduction in L1 data cache read misses per instruction executed. The benchmark program with the second largest reduction in L1 data cache read misses, *MG3D*, had a large part of the benefit of this reduction cancelled out by an increase in the L2 data cache misses, as discussed already in relation to Table 5-7. Note also that *FLO52*, on a per-instruction basis, had almost as much reduction in L2 cache data misses as in L1 data read misses. Because the miss penalty is several times higher for the L2 cache, a large percentage of the speedup of *FLO52* came from L2 improvements. This is the only program for which this seems to be the case.

Another measurement that influences the changes per instruction executed is the first-level data cache read miss rate. If the miss rate is high before data placement, then there is potential for enough improvement to produce a high speedup. If there is a low miss rate prior to data placement, then there is little potential for data placement to accomplish s high speedup, even if it is effective in reducing the cache conflicts. Table 5-10 shows the before and after miss rates for first-level data cache reads, and shows the improvement as

a simple difference of these two rates. The difference as a percentage of executed instruc-

Program	L1D read miss rate before placement	L1D read miss rate after placement	Delta
compact	1.680%	0.102%	1.578%
ADM	3.718%	2.692%	1.026%
ARC2D	12.193%	11.720%	0.473%
BDNA	10.474%	6.572%	3.902%
DYFESM	2.005%	1.170%	0.835%
FLO52	5.247%	4.904%	0.343%
MDG	2.211%	1.329%	0.882%
MG3D	4.560%	3.363%	1.197%
SPEC77	3.924%	3.546%	0.378%
TRACK	3.426%	3.166%	0.260%
TRFD	2.539%	2.122%	0.417%

TABLE 5-10: First-level data cache read miss rates.

tions was already given in Table 5-9.

As shown in Table 5-10, it is no surprise that the speedup was greatest for *BDNA* among the Perfect Club benchmark programs. Six of the other nine Perfect Club programs had a lower cache miss rate before placement than the *reduction* in cache miss rate achieved for *BDNA*, meaning that even reducing the cache misses to zero for these programs would have produced a smaller absolute reduction in miss rate percentage than was actually produced for *BDNA*. The programs that seem to offer significant potential for speedup other than *BDNA* are *ARC2D*, *FLO52* and *MG3D*. The latter two in fact had the second and third highest speedups. *ARC2D* suffers from having less than 1% of its data in the first (smallest) category and more than 50% in the third (largest) category, as shown in Table 5-2.

In order to further test the hypothesis that accidental conflicts between code lines and data lines in the second-level unified cache are having a significant effect on some of the benchmark programs, cache simulations were run in which the second level cache was split into a data cache and an instruction cache. Because the data placement optimizations perform calculations using the full size of the second-level cache, each cache was specified

as a direct-mapped 128KB cache with 64-byte lines. The size of the instruction cache is actually irrelevant here. Because the data placement optimizations do not change the instruction stream, the before-placement and after-placement versions of a program will have identical behavior in the split instruction caches, both first-level and second-level. This is confirmed by the cache simulation data. The hypothesis is that some programs have significant back invalidations in the first-level data cache that are caused by conflicts of code and data in the second-level cache. Because these conflicts will be dealt with in follow-on work to this research, insight into the effectiveness of the current research can be gained by removing those conflicts and measuring the results. On the contrary, if the back invalidations in the first-level data cache are caused by data cache conflicts alone, then this will be revealed by a split-cache simulation.

Table 5-11 shows the first-level data cache read miss reductions for the unified cache and split cache simulations. The unified cache data are copied from the last column

Program	Change in L1 data cache read misses (unified)	Change in L1 data cache read misses (split)	Delta
compact	-93.91%	-93.91%	0.00%
ADM	-27.59%	-22.40%	+5.19%
ARC2D	-3.88%	-3.93%	-0.05%
BDNA	-37.25%	-38.44%	-1.19%
DYFESM	-41.65%	-51.41%	-9.76%
FLO52	-6.52%	-6.50%	+0.02%
MDG	-39.86%	-39.87%	-0.01%
MG3D	-26.24%	-26.61%	-0.37%
SPEC77	-9.62%	-8.19%	+1.43%
TRACK	-7.60%	-15.32%	-7.72%
TRFD	-16.40%	-16.40%	0.00%

TABLE 5-11: Effect of using split second-level caches.

of Table 5-4. Both the before-placement and after-placement versions of the program benefit from removing all code from the second-level cache. Rather than show the benefits directly, the focus is on whether the after-placement version of the program benefits more

or less than the before-placement version. The final column shows the difference in the reduction percentages, i.e. a simple difference of the previous two columns, shown as the first column subtracted from the second column to remain consistent with previous tables (i.e. a negative value is good, showing improvement through increased reduction of cache misses). Seven of the delta values are quite small, indicating that there was little effect on the first-level cache due to splitting the second-level caches. The program *ADM* actually benefits in the first-level data cache from the unified cache relative to the split cache. This is because there was a degradation in second-level cache performance, caused by code versus data conflicts, that was improved by the smoothing effect of data placement. When the code versus data conflicts are removed by splitting the second-level caches, there is less relative benefit from data placement because only the data conflicts can now be improved by placement. For *DYFESM* and *TRACK*, the opposite effect occurred. The data placement effectiveness was reduced by code versus data conflicts that caused back invalidations in the first-level cache, as previously predicted. Using a split cache, even though it reduced the first-level data cache misses in the before-placement version of the program, led to an even greater reduction in the after-placement version of the program, because the back invalidations were a bigger problem in that version.

An interesting fact to note from Table 5-11 is that *DYFESM* achieved the highest percentage reduction among the Perfect Club benchmark programs in first-level data cache read misses for both unified and split second-level caches, yet its speedup was fourth best among the Perfect Club programs and was only about one third the speedup achieved for *BDNA*. As seen in Table 5-10, there was no possible means by which the speedup of DYFESM could match that of *BDNA*, even if there were no code versus data conflicts at all, because of the difference in cache miss rates prior to placement. It can be accurately stated that the data placement optimization was more effective for *DYFESM* than for *BDNA*, probably due to the different distribution of data sizes between the programs, as seen in Table 5-2, yet the speedup was far less. This demonstrates the value of detailed examination of cache simulations for understanding program behavior and optimization effectiveness, rather than relying on timings alone. It also demonstrates once again that larger input data sets would provide more opportunity for speedup through data placement optimizations.

The high effectiveness of data placement for *DYFESM* with the split second-level caches motivates the following question: How effective is the data placement optimization, on programs where it seems to be most effective, in comparison to increasing the associativity of the first-level data cache? Both approaches, one in software and one in hardware, seek to reduce cache conflicts. The software approach consumes some compilation time (which will be discussed in the next section), while the hardware approach has costs in both cycle time and power consumption in cache tags as the cache associativity increases [Jouppi and Wilton 1994]. Cache simulations using the split second-level cache model for *DYFESM* can provide a comparison of first-level data cache read misses and rates at various levels of associativity, as shown in Table 5-12. By comparison, data placement, with

Sets	L1D read misses without placement	L1D read miss rate without placement
1	20,639,110	2.011%
2	11,967,169	1.166%
4	12,136,019	1.183%
8	12.768,089	1.244%
8 (LRU)	11,688,671	1.139%
16	13,055,005	1.272%
16 (LRU)	9,484,856	0.924%
512*	11,157,906	1.087%
512* (LRU)	9,148,842	0.892%

TABLE 5-12: Associativity effect on *DYFESM* data cache misses.

the split second-level cache, has only 9,763,657 misses, for a miss rate of only 0.951%.

The asterisk next to the 512-way associative entries indicates that these simulations used a 4-way-associative second-level split data cache to reduce back invalidations to a near-zero level. Also, 512 sets is fully associative, so these entries are close to the theoretical limit of what could be achieved with hardware changes to the cache. All data is for a random replacement scheme, unless the entry specifies LRU (least recently used). Random replacement incurs *replacement misses*, which are misses caused by a previous cache eviction of a cache line that was not (in hindsight) the best choice out of its set to evict, because

it was going to be reused sooner in the future than at least one other available choice. The effect of replacement misses can be seen in the degradation of performance with increased associativity among the entries using random replacement, starting with the 4-way entry.

Reducing replacement misses with LRU replacement is increasingly expensive as the associativity increases. For a 16-way associative cache, for example, a complete ordering of the sixteen items in a set must be maintained at all times. There are 16! (16 factorial) possible orderings, so pseudo-LRU approximations are almost always used in practice, rather than true LRU, even starting with 4-way associative caches.

With either random or LRU replacement, hardware changes are not able to match the effectiveness of data placement in reducing first-level cache misses until the cache configuration has become much more expensive than the original direct-mapped cache. While *DYFESM* is apparently the best such example among the Perfect Club programs, it is still a revealing insight into the effectiveness of the data affinity analysis and data placement.

Although no effort in the analysis or placement attempted to reduce first-level data cache write misses, because the first-level data cache has a policy of write-through, no write allocate on miss, it could be expected that increased locality and reduced conflict on reads would cause write misses to be reduced as a side effect. To indirectly test the effectiveness of the data placement framework in increasing locality of data cache writes, which could have a side effect on page locality and data TLB miss rates (writes and reads are equally significant in paging and TLB access), the reduction in first-level data cache write misses is shown in Table 5-13.

The hierarchical partitioning design, followed by confining the local refinement stage to rearrangements within partitions and not across partitions, should be expected to improve page locality and reduce data TLB misses. By using the *cachesim5* cache simulator with a first-level data cache that mimics the data TLB parameters (e.g. 8192 byte line size, which equals the page size; 512KB capacity; 64-way associativity with random replacement), the data TLB misses for an UltraSparc-IIi system can be measured. The results of these measurements are in Table 5-14. The nearly uniform improvements are another validation of the effectiveness of the multi-goal data placement framework. Because all of the global data for compact fits in only three 8KB pages, which means that the only data TLB misses should be the first three compulsory misses regardless of data

Program	L1D write misses before placement	L1D write misses after placement	Change
compact	64,261,014	10,075,430	-84.32%
ADM	25,557,259	16,282,588	-36.29%
ARC2D	292,176,704	268,292,191	-8.17%
BDNA	78,264,631	40,172,359	-48.67%
DYFESM	4,386,835	3,831,181	-12.67%
FLO52	75,152,217	74,252,984	-1.20%
MDG	63,335,416	13,549,279	-78.61%
MG3D	2,389,169,573	2,349,950,454	-1.64%
SPEC77	323,813,031	322,587,728	-0.38%
TRACK	7,854,293	7,146,694	-9.01%
TRFD	6,374,568	5,976,805	-6.24%

TABLE 5-13: First-level cache write misses.

Program	Before placement	After placement	Change
ADM	222	162	-27.03%
ARC2D	82,197,127	81,330,149	-1.05%
BDNA	153,388	156,127	+1.79%
DYFESM	171	150	-12.28%
FLO52	87,553	76,136	-13.04%
MDG	248	198	-20.16%
MG3D	512,865	407,756	-20.49%
SPEC77	208,268	196,392	-5.70%
TRACK	6,667	6,276	-5.86%
TRFD	886,308	886,128	-0.02%

TABLE 5-14: Data TLB misses.

placement reorderings within those three pages, measurements were not made on *compact*. The only slight degradation in data TLB misses was for *BDNA*. As noted previously, it seems from the cache simulations that *BDNA* suffered a small performance degradation in the second-level cache due to the interactions among the larger arrays and the smaller arrays, which would require array subscript analysis to avoid. The data TLB maps 512KB

of data, while the second-level data cache holds 256KB, so issues involving data of a certain large size will have similar effects on the second-level cache and the data TLB. The primary difference between the two elements of the memory hierarchy is that the TLBs on the UltraSparc-IIi are not unified, so the code versus data conflicts that were a significant issue on several benchmark programs do not affect the measured data TLB misses.

5.2.1 Data Placement Of Local Variables

None of the benchmark programs measured has a large enough stack frame in any function to require placement of locals within stack frame. Such large stack frames are rare. In the SPEC CPU2000 benchmark suite, the program *177.mesa* had four such stack frames, including a heavily-used function with a 36KB stack frame. The combination of data placement and addressing range optimizations reduced execution time by 2%, with half the improvement coming from reducing cache misses and the other half from reducing the count of instructions executed. These results were obtained with a version of the static analysis and placement framework that has become obsolete. Recent improvements in the global data placement framework need to be propagated to the local placement framework before definitive measurements can be made.

5.3 Analyzing the Optimization Stages

In this section, data concerning the stages of the data placement framework are analyzed to gain insight into the costs and benefits of each stage.

The first stage examined is the static data affinity analysis. The effectiveness and correctness of the analysis of accesses to global variables and formal parameters was verified through many hours examining debugging output from within *vpo* that was switched on and off as development progressed through the different stages of the research. In addition, many hours were spent in a debugger examining the operation of the analysis step by step. Upon completion, there were no known cases in the benchmark programs examined of failure to profile an access to a global variable or formal parameter, nor any failures to match actual parameters to their corresponding formal parameters. The static data affinity

analysis is a key foundation of the research, upon which the placement effectiveness depends, and easily received the most time and attention of any stage.

One statistic gathered was the reason for the termination of each path in the path analysis step. These numbers are presented in Table 5-15.

Program	Size	Freq.	Rep.	Loop
compact	0	68	7	0
ADM	957	2	0	4471
ARC2D	613	593	393	84
BDNA	11	176	0	1002
DYFESM	0	148	0	1568
FLO52	531	279	3247	10
MDG	0	46	1	298
MG3D	1	8	2	6617
SPEC77	1041	27884	1808	607
TRACK	0	3448	296	266
TRFD	65	75	110	0

TABLE 5-15: Path analysis path terminations by cause.

Recalling the discussion of path analysis from Chapter 3, the four categories in the order shown above are: reaching a capacity limit (currently set at twice the size of the second-level cache); reaching a CFG node with an execution frequency estimate below 1.2; seeing a repetition of each global variable from the head node of the path somewhere in the path, which means that the variable is reloaded in the cache and no other variable will evict the instance found in the head node; and reaching a node that does not share any loop in the loop nests with the head node of the path. If a program's path terminations are significantly affected by the *size* category, that would indicate a program structure with large data items being accessed frequently. A significant effect from the *frequency* category indicates that there are many outermost loops in succession, or much branching within outer loops. A large effect from the *repetitions* category indicates high reuse of data within loops. Finally, a high termination rate from the *loop* nest category indicates a program with many different loop nests in succession.

With the size cutoff set at twice the second-level cache size, meaning 512KB, the size cutoff count should be zero for programs with less than 512KB of data. Only *compact*, *DYFESM* and *MGD* fit this criterion, and their size cutoff counts are indeed seen to be zero. Verification that the other path cutoff counts match the program structure can only be accomplished by inspecting the source code; at this time, no discrepancies have been seen.

The packing cache lines stage can be measured using the number of cache lines packed and the sum of mutual affinities packed into the line (which was the performance criterion for this optimization). These numbers are shown in Table 5-16.

Program	Affinity	Lines	Density
compact	1207	6	201
ADM	930,193,944	203	4,582,236
ARC2D	26,269,754	120	218,915
BDNA	29,140,884	212	137,457
DYFESM	134,135,527	147	912,487
FLO52	6,856,815,766	94	72,944,849
MDG	561,638	71	7,910
MG3D	11,751,554,682	63	186,532,614
SPEC77	30,233,598,179	138	219,084,045
TRACK	3,132,625	98	31,966
TRFD	27,366,329	29	943,667

TABLE 5-16: Cache line packing data.

The key measurements that cannot be obtained are the total affinity within cache lines before and after packing the cache lines, and the effect of cache line packing on the number of cache lines in the working set size of the program. The former number would measure affinity across all elements within a line before cache line packing, including affinity between a scalar at the beginning of a line and the early elements of an array that begins in the middle of the line, for example. This would actually require array subscript analysis to be done precisely.

After cache line packing, the data placement continues with hierarchical graph partitioning using *Chaco*, and then an iterated local refinement stage using swaps and rota-

tions. Before *Chaco* is first called, the cache map is built and the initial worst-case cache conflict metric is computed cache line by cache line, as described in Chapter 4. After *Chaco* has partitioned the global data into second-level-cache-size partitions, and then into first-level-cache-size partitions, the cache map is rebuilt and the conflict metric is recalculated. Then, up to four iterations of the local refinement stage are made, with the cache conflict metric recomputed continuously throughout the process in order to accept beneficial changes and reject any swap or rotation that is harmful or neutral in effect. Using debugging switches, this data is printed so that the relative importance of these steps can be examined. Table 5-17 breaks down the percentage of the reduction in the conflict metric that was achieved by hierarchical graph partitioning using *Chaco*, and the percentage achieved by the local refinement iterations. The two numbers will sum to 100% in each case.

Program	Partitioning	Refinement
compact	N/A	N/A
ADM	N/A	N/A
ARC2D	62%	38%
BDNA	21%	79%
DYFESM	89%	11%
FLO52	26%	74%
MDG	99%	1%
MG3D	95%	5%
SPEC77	36%	64%
TRACK	87%	13%
TRFD	N/A	N/A

TABLE 5-17: Relative conflict reductions by optimization stage.

Note that the cache map cannot be built until the cache line packing stage has been completed. The granularity of the cache map is the size of the second-level cache lines, which means 64 bytes for the UltraSparc-IIi machines. A pointer to the leftmost scalar in the packed line is stored in the cache map entry for that line, and all affinity is attributed to that leftmost scalar after cache line packing. Before packing, a cache line could have several scalars followed by the beginning of an array, and a single global symbol table pointer

could not represent the cache line. Thus, measurements of the reduction in the cache conflict metric achieved by the cache line packing stage are not possible.

The entries marked as not applicable are because some programs have a conflict metric of zero after cache line packing. This was discussed as a special case for *compact*, while *ADM* and *TRFD* have so little data that is smaller than the second-level cache size that there is no conflict and thus no need for the partitioning and refinement stages.

The numbers demonstrate the power of both stages. On several programs, graph partitioning accomplishes almost all of the reduction in cache conflict. This is true for programs in which little reduction is accomplished, such as *MDG*, but also for *DYFESM*, which had the highest percentage reduction of L1 data cache read misses of all Perfect Club programs. However, local refinement is obviously crucial to the results of about half the programs, and was most significant for *BDNA*, which was the Perfect Club program with the highest speedup.

Most of the benefit from the local refinement stage consists in finding a few swaps that each have a large benefit. These swaps are usually found in the first iteration of the local refinement procedure over the global data partitions. Rotations rarely account for significant reductions in the conflict metric. A summary of the relative contributions of the five best swaps and all beneficial rotations is given in Table 5-18.

Program	Top 5 swaps as percent of local refinement benefit	Rotations as percent of local refinement benefit
ADM	N/A	N/A
ARC2D	96.8%	0.0%
BDNA	79.2%	1.2%
DYFESM	96.3%	2.8%
FLO52	37.1%	57.3%
MDG	0.0%	100.0%
MG3D	59.1%	32.6%
SPEC77	53.6%	37.2%
TRACK	77.8%	14.7%
TRFD	N/A	N/A

TABLE 5-18: Relative benefits of the best five swaps and all rotations.

The *MDG* benchmark receives the benefit of one good rotation and zero good swaps, producing the inflated value for the rotations. MDG has less total data than the size of the second-level cache, and partitioning accomplishes all of the reduction in first-level cache conflict that can be found outside of the tiny benefit from the single rotation. Experience with the other beneficial rotations indicates that they are mostly surrogate swaps; i.e., a single area of conflict is improved by the rotation, and that same area receives the same benefit from a swap when rotations are turned off. The designed purpose of rotations was to help avoid local optima by striking out into the search space to a point not reachable by a swap. In practice, rotations seem to be accomplishing little in that regard. This could be interpreted positively to mean that the data placement optimization is not getting stuck in local optima, as predicted in Chapter 4. There, it was noted that if a particular partition is stuck in a local optimum, the other partitions that conflict with it can still perform their own swaps, so the placement optimization is not necessarily stuck in a local optimum within the global search space over all global variables.

The compilation time increase due to the data placement framework can be measured in two ways. In order to perform the analyses and optimization of this research, the *vlink* tool must be used to collect all of the intermediate code files of the program into a single merged file. Then a call graph building pass must be made over all RTLs in the program, during which some of the data affinity analysis work is done. At the conclusion of the call graph processing, the inlining of CFGs, path analysis, cache line packing, partitioning, and local refinement of partitions are all performed. Then, the input file is rewound, and *vpo* begins its normal processing.

If the time from the *vlink* processing and call graph pass, which are essential to this research, were counted as part of the compilation cost of the research, then the compilation cost would be about three times as high as it would be if those costs were not included. The *vlink* and call graph stages will be required by many interprocedural optimizations in the future. Currently, only the present research and the memory hierarchy research of Jason Hiser require these steps, but the cost of these steps will be amortized over a growing number of optimizations in the coming years.

Comparing the total compilation time for all data placement optimizations in this research to a baseline that includes the *vlink* and call graph stages, the change in compilation times can be seen in Table 5-19. All compile times are in seconds.

Program	Compile time w/o placement	Compile time with placement	Change
compact	19	20	+5.3%
ADM	3794	4918	+29.6%
ARC2D	1107	10165	+818.2%
BDNA	1932	3723	+92.7%
DYFESM	607	706	+16.3%
FLO52	806	2149	+166.6%
MDG	196	219	+11.7%
MG3D	946	2117	+123.8%
SPEC77	1149	6232	+442.4%
TRACK	304	917	+201.6%
TRFD	55	56	+1.8%

TABLE 5-19: Compilation times in seconds.

A few of the programs in Table 5-19 have artificially low increases in compilation time because they have no conflict after cache line packing, and hence do not invoke partitioning and local refinement at all. As mentioned previously, these programs are *compact*, *ADM*, *MDG*, and *TRFD*. A few of the bigger programs dominate the compilation time of the benchmark suite as a whole, such as *ARC2D* and *SPEC77*. The local refinement stage is the primary consumer of compilation time, and there are several straightforward means that can be employed to reduce its time consumption as follow-on work. The easiest speedup will come from reducing the iterations of local refinement from four to two for most programs. The current limit is four, with a premature termination if two consecutive iterations produce less than one percent reduction in the cache conflict metric. It is observed in practice that, after a single iteration produces a negligible benefit, no further iterations will produce non-negligible benefits. By defining a negligible benefit as a two percent reduction in the cache conflict metric, rather than the current one percent, and terminating the local refinement iterations after a single iteration produces a negligible benefit, a sig-

nificant reduction in compilation time will be realized with negligible impact on the effectiveness of the optimizations. This change has not yet been implemented, because proving that the effectiveness has not been significantly reduced will require new runs of numerous lengthy cache simulations.

The other change to the local refinement to reduce compilation times will be guaranteed to not affect the optimization results. A large portion of the local refinement time is spent recalculating the cache conflict metric after every swap and every rotation in order to accept or reject the transformation. The conflict metric is stored in an array dimensioned by the number of cache lines in the cache map. By only recalculating the conflict metric array entries that will be affected by a swap, the recalculation time can be reduced greatly. Rotations affect the entire conflict metric array, so this idea only applies to swaps.

The compilation time consumed by various stages in the data placement framework are shown in Table 5-20 and Table 5-21. The inlining and analysis times are good indicators

Program	CFG Inlining	Path Analysis	Cache Line Packing
ADM	4	414	671
ARC2D	1	8	59
BDNA	1	146	222
DYFESM	1	5	80
FLO52	1	13	37
MDG	1	2	13
MG3D	3	242	47
SPEC77	2	315	126
TRACK	1	35	40
TRFD	1	1	2

TABLE 5-20: Compilation time in seconds by stage.

of the relative code sizes, while the remaining times are good indicators of the relative data sizes. Inlining CFGs is the stage that is most heavily fine-tuned for fast compilation.

Program	Partitioning	Refinement	Refinement iterations
ADM	N/A	N/A	N/A
ARC2D	27	10153	3
BDNA	6	2737	4
DYFESM	1	9	3
FLO52	35	1309	4
MDG	1	8	2
MG3D	8	1156	3
SPEC77	566	3811	4
TRACK	11	404	3
TRFD	N/A	N/A	N/A

TABLE 5-21: Compilation time in seconds by stage (continued).

5.4 Comparison to Prior Work

The most comprehensive data placement optimization research prior to this work was conducted by Calder, Austin et al. [Calder et al. 1998], combining the best features of the code placement framework developed by Hashemi, Kaeli and Calder [Hashemi et al. 1997] with the data placement framework developed in Austin's dissertation [Austin 1996]. This framework included a custom heap allocator, heap and stack profiling along with global profiling, and the expensive but accurate TRG method of dynamic profiling that was discussed in Chapter 2. This work is less comprehensive than the present research in that it has the single goal of reducing first-level data cache misses, but it is more comprehensive in that it profiles and places heap data items, permitting use of six of the SPEC95 benchmarks, along with two C++ programs. An impressive arithmetic average of 23.75% reduction in first-level data cache misses (16.06% geometric mean) was measured across nine benchmarks for an 8KB direct-mapped data cache using a cache simulator for the ATOM tool [Srivastava and Eustace 1994].

Because of the single-goal nature of the placement framework, there was a slight degradation in paging and TLB performance. All results were presented from the SimpleScalar simulator in terms of cache misses and pages used, with no timings. Some cache line

packing was performed, but the algorithm used was not described. As with the present research, no array subscript analysis was available. Related optimizations such as addressing range reduction and access coalescing were not performed. Levels of cache beyond the first-level data cache were not targeted. An extension of their algorithm to associative caches was reportedly in development at the time they published the CCDP results in 1998, but has not yet been published. The CCDP approach is somewhat greedy and local and does not suggest easy extension to multilevel cache hierarchies, differing degrees of associativity, increased page locality, etc. Its dependence on the ATOM tool for editing executable code reduces its portability, as ATOM works only on Alpha CPU systems [Srivastava and Eustace 1994].

(It should be noted that general comparisons can be made between the characteristics of CCDP and multi-goal data placement, but a direct comparison of effectiveness is not possible due to the different benchmark programs measured in each work. The SPEC95 and C++ benchmark programs optimized by CCDP have characteristics that are both beneficial and harmful when used to demonstrate the effectiveness of a data placement framework. All comparisons below are intended to give insight into the effectiveness of multi-goal data placement and do not constitute a proof of its superiority to CCDP.)

In contrast, the present research does not perform analysis or placement of heap objects, and the Perfect Club benchmarks that were chosen as a result have much shorter run times than the SPEC95 benchmarks that were available to Calder et al. The average data cache miss rate prior to their Cache Conscious Data Placement (CCDP) optimization was 9.03%, which was reduced by CCDP to 7.50% on average. The percentage point reduction in cache miss rate was therefore 1.53 on average (1.30 using the geometric mean), with a range from 0.00 to 3.67. This compares with a range of 0.26 to 3.90 (average 1.02) in the present research, despite much lower prior miss rates.

If timings had been available, it seems likely that their execution time speedup would have ranged from 1.000 up to 1.065. This compares to the range of 1.011 up to 1.070 for the present research, despite the much lower rate at which the data is exercised in the Perfect Club benchmarks as compared to the benchmarks used by Calder et al. The mean speedup of the CCDP work on their benchmark programs would probably exceed the mean speedup of this research on Perfect Club programs; note that the 1.070 speedup on BDNA

is twice as good as the speedup of the second-best Perfect Club code. The average first-level data cache read miss reduction is 28.23% for the present research, compared to 23.75% for CCDP (19.16% compared to 16.06% using geometric means).

The comparison between the present research and CCDP is summarized in Table 5-22, where the present research is referred to as Multi-goal placement. Mean values are specified as arithmetic or geometric.

Criterion	CCDP	Multi-goal placement
L1D miss reduction (arithmetic)	23.75%	28.23%
L1D miss reduction (geometric)	16.06%	19.16%
Prior miss rate (arithmetic)	9.03%	4.73%
New miss rate (arithmetic)	7.50%	3.70%
Prior miss rate (geometric)	8.06%	3.87%
New miss rate (geometric)	5.41%	2.36%
Speedup range (est.)	1.00 to 1.065	1.011 to 1.070

TABLE 5-22: Comparison with CCDP (different benchmarks used).

While it is encouraging to see that the present multi-goal data placement achieves greater cache miss reductions than prior work, even if direct comparisons on the same benchmark programs are not available, these numbers are only a part of the advancement in data placement represented by this research. Dynamic profiling is always supposed to be superior to static analysis [Patterson 1995], which should give a cache miss reduction advantage to CCDP at the cost of an expensive profiling run. Numerous prior efforts in both code and data placement have had no practical extension to associative caches [Kalamatianos 2000], [Calder et al. 1998], [Austin 1996], [Hashemi et al. 1997]. Multi-goal data placement uses a cache map with an conflict metric calculation that easily takes associativity into account during the local refinement stage. The graph partitioning stage has no need to consider associativity. Prior data placement work has not targeted a multi-level cache hierarchy, and there has been only a single unsuccessful attempt to consider multiple levels

of cache in a code placement optimization [Kalamatianos 2000]. Hierarchical graph partitioning is intuitively adaptable to any number of cache levels. Data affinity analysis and local refinement need only maintain separate affinity arrays for each level of cache in order to effectively target a multi-level cache hierarchy. Hierarchical graph partitioning, as refined through the fine tuning of parameters within *Chaco*, can be made to increase page locality and reduce TLB misses, an area of the memory hierarchy that has not been managed well in past work. Finally, unified cache code and data conflict reduction is facilitated naturally by the unique static data affinity analysis contribution of this work.

In summary, the superiority of the techniques employed in multi-goal data placement is demonstrated even more through its adaptability to numerous aspects of real machine architectures and microarchitectures than through its measurable improvements in cache miss reduction.

5.5 Enhancements Suggested by the Measurements and Results

The first improvement suggested by these measurements is the acquisition of benchmarks with larger input data sets and higher cache miss rates, in order to translate the excellent cache miss reductions into higher speedups.

The second improvement suggested by these measurements is the integration of a code placement optimization in order to reduce the negative effects of back invalidations.

The third improvement suggested is the integration of prior work in array subscript analysis, in order to make the data placement optimizations more effective on programs whose global data is concentrated in one or a few very large arrays.

Finally, the range of benchmarks studied could be improved by analyzing and placing heap data.

Chapter 6

Determining Cache Parameters and Retargeting the Optimizations

Details of the cache hierarchy have determined decisions made in the static data analysis and the optimizations. For example, static data analysis had a cache capacity cutoff, among other path termination criteria, implying that the sizes of first and second level caches are known. Whether to profile loads and stores, or just loads, for a particular level of cache depends on the write-miss-allocation policy of the caches. How to scale and proportion global variable affinities along a profiled path so that they reflect worst-case cache evictions requires knowledge of line and subblock sizes for all levels of cache. Cache line packing requires knowledge of the line size of the second-level cache. Local refinement of the data placement requires a cache map and a cache conflict metric computed over that cache map, which requires knowledge of the associativity and total size of each level of cache. The cache conflict metric requires an estimate of the ratio of the second-level cache miss penalty to the first-level cache miss penalty.

In summary, for the methods described in this research to function properly, the following information is needed:

- **Cache line (block) and subblock sizes for all cache levels.**
- **Cache total size for all cache levels.**
- **Cache associativity for all cache levels.**
- **Cache write-miss policy for all cache levels.**
- **Ratios of miss penalties for each cache level compared to the first cache level.**

In addition to this data, the load coalescing and address-range optimizations require further information about the target machine, which will be described at the end of this chapter.

How to obtain all of this information is the subject of this chapter.

6.1 Background

Characterizing the cache and memory hierarchy of a computer is of broad interest. Because the gap between processor performance and memory performance continues to grow, performance analysis and compiler optimizations are increasingly focused on the memory hierarchy of the target computers. At first, it might seem that such information could be easily obtained from computer vendor documents. While this is sometimes the case for certain parameters of the memory hierarchy, there are numerous deficiencies in this method, based on experience:

1. Vendor documents can be incorrect. For example, the Intel P6 Family Instruction Set Architecture Manual ([Intel Corporation 1997]; see page 3-74), in its description of the CPUID instruction, describes the 16KB L1 (Level One) data cache associativity as being 2-way when it is in fact 4-way. Later corrections to this value and several TLB values in this manual confirm the presence of errors over time.
2. Vendor documents can be vague, such as when a single manual attempts to describe several related processors within a family. This often leads to describing the memory hierarchy with a range of possibilities, or only with the common denominator parameters.
3. Vendor manuals often describe memory hierarchy components that are used by the operating system but are not accessible to user-level code under that operating system ([MIPS Technologies 1996]; compare sections 14.6 and 16.3 on *wired TLB entries*), such as the large-page TLBs in the Intel P6 family processors [Intel Corporation 1997].
4. Some parameters are dependent on the system, not the CPU, and thus are not documented fully in CPU manuals. For example, off-chip secondary cache sizes typically vary in powers of two among systems using a certain CPU. The size is best determined dynamically.
5. Gathering information through searches of processor, system, and OS documents, followed by email correspondence to resolve ambiguities and errors, is very time-consuming.

For these reasons, documentation is an insufficient source for needed memory hierarchy information. A superior approach would be to design a program that can reliably perform dynamic memory hierarchy characterization directly on the machines to be characterized. Such a program would reveal the memory hierarchy as actually seen by the application programs on the system and would not include memory hierarchy elements that are visible only to the operating system. As part of this research, such a program has been successfully designed and tested, which is described in this chapter. Because this program was designed for use by a wide community of researchers, and not just for use in the present research, it determines many parameters of the memory hierarchy beyond those required for this research. A description of the program has been published previously [Coleman and Davidson 2001].

6.1.1 Dynamic Measurements Using Timing Tools

Prior tools have measured latency, bandwidth, and cache sizes using the timing precision available in the standard Unix/C environment ([McVoy and Staelin 1996], [Brown and Seltzer 1997]). Through repeated memory accesses (in which lengthy straight-line code repeatedly dereferences and then increments a pointer) and timings, these programs can detect the slowdown that occurs when a data array exceeds the size of a level of cache, and hence can determine the size of that level of data (or unified) cache, and its latency and bandwidth. These tools are valid and useful for their intended application.

There are limitations in this approach, however. Existing tools do not measure instruction caches, nor instruction or data TLBs. Thus, they do not give a complete, precise measurement of the parameters required for optimal use of the memory hierarchy.

To address the problem of gathering accurate, reliable data about the characteristics of all aspects of the memory hierarchy, we have designed a portable program, called AMP (for Automatic Memory hierarchy Parameterization), that automatically determines all key memory hierarchy characteristics including those of the instruction caches and TLBs. Most modern processors incorporate hardware performance counters that can count certain events, such as cache or TLB misses. However, there are several challenges to using such counters.

First, there is no common set of counters available across all processors. The counters available and what events are recorded vary widely even among processors from the same vendor. Second, accessing the counters that are available is machine and operating system dependent. To overcome these challenges, we have designed a middle layer module that acts as an interface between AMP and the application program interface (API) for accessing the counters. There are currently several research groups providing APIs for access to these performance counters. The APIs germane to this project are described in Section 6.2.1.

A third, and perhaps most difficult challenge, is that it is not feasible or reasonable to run measurement programs such as AMP on a standalone machine. For example, many systems require connections to the network for file system access. Configuring a machine to run in a standalone fashion is often not an option (e.g., removing a compute server from

the pool of available servers), and when it is an option, taking a machine and rebooting it in standalone mode is time-consuming. To address this problem, we designed AMP so it can be run on a machine in a standard networked environment. By carefully controlling the experiments that are run and using statistical techniques, AMP accurately determines the memory hierarchy parameters on a machine being run in a standard computing environment.

The remainder of the chapter has the following organization. In the next section, we address issues regarding portability and the robustness of our measurements on multi-tasking systems. Section 6.3 describes the algorithms developed to compute the various memory system parameters. Section 6.4 discusses the measurements collected for a variety of systems, Sections 6.5 and 6.6 describe the implementation and availability of the software, and Sections 6.7 and 6.8 summarize the supplements to the work and future enhancements. Finally, the work required to retarget the full body of this research, including data affinity analysis and data placement related optimizations, is listed in Section 6.9.

6.2 Robustness and Portability

In this section, we address issues confronted when running AMP on multiple targets that have different performance counters available, with different access methods. We also show how we solved the problems introduced by competition on multi-tasking systems.

6.2.1 Performance Counter APIs and Portability

AMP currently runs primarily on top of the PCL performance counter API ([Berrendorf and Ziegler 2003]), available on several hardware platforms. A similar API effort is PerfAPI, also known as PAPI ([Mucci et al. 1999]). The Rabbit API is available for x86/Linux measurements only ([Heller 1999]). AMP has been ported to these APIs. Development work was undertaken in the DOS/DJGPP environment ([Delorie 2001]), which permitted direct counter access in privileged mode under DOS, without multitasking overhead.

We have created a middle layer module that acts as an interface between the code implementing the algorithms described above and the performance counter API. Queries (available in all three APIs) are used to determine what events are available on a system

(and thus, how many levels of cache have associated counters). Porting to a new performance counter API only requires changes to this middle layer module.

A machine-dependent module can use machine-specific instructions to flush caches, TLBs, etc., but can be made to just return `false` for all functions to minimize porting effort. When a return value of `false` is seen, the cache flushing proceeds as described in Section 6.3.1, Step 1, in which code or data accesses are used to evict from the cache the code or data of interest.

6.2.2 Robustness and Reliability

We observed during testing that AMP suffered a loss of reliability whenever the APIs (or competing system processes) caused cache misses unrelated to AMP's algorithms. A voting scheme eliminated this problem. The main program actually gathers measurements in a voting loop. An outline of the measurement and voting approach is as follows.

```
1. while (vote limit not reached) do
2.   Perform experiment measuring cache
     misses 8 times
3.   Take minimum of these 8 numbers
4.   Use the minimum to compute the
     result (e.g. line size)
5.   Store result as the current vote
6. endwhile
7. The final value = most common vote
```

Using the minimum misses from multiple repetitions discards high numbers of misses caused by context switching or other anomalies. Our hypothesis is that no event in the system will reduce measured cache misses, but many events could increase them, so the minimum is the most accurate measurement. (Note that the APIs used count misses on a per-process basis; the problem is not the accidental counting of misses suffered by another process, rather the increase in AMP process cache misses caused by cache evictions that occur during execution of another process, causing certain data items to not be found in the cache as expected.) Thus, computing the L1 D-cache line size involves taking eight measurements, recording the minimum number of misses among those eight, using that minimum to compute a line size, and making that line size the first vote, etc. An important point here is that all measurements are designed to produce hundreds of cache misses, or at

least dozens of TLB misses; AMP does not depend on a precise number of misses that is small, as this would be unreliable even after implementing this voting scheme. In all, 64 measurements will produce 8 reliable votes. Prior to implementing the voting scheme, AMP produced inconsistent results on different runs on very lightly loaded Unix systems. With voting, AMP returns the same parameters on more than 90% of all runs.

Another boost to our measurement reliability is the computation of the performance counter API overhead. The API might make small cache perturbations, which need to be removed from our measurements. At startup, AMP loops through an array that has been read into the cache and should be small enough to remain there for a brief duration. AMP counts the cache misses (which would be zero, if there were no API overhead) and takes the minimum of several counts as the overhead for that measurement. This is done for all caches and TLBs. The overhead is subtracted from each measurement described in this paper before any cache miss number is used in determining any cache parameter.

6.3 Algorithms and Measurements

The subsections below describe our measurement algorithms in terms of performance counters while making reference to particular machine dependencies only where necessary. Keep in mind when reading these measurement algorithms that each measurement is actually a sequence of repeated measurements, followed by a voting scheme, as discussed in Section 6.2.2.

In the data cache discussions, AMP uses a very large integer array declared as a C-language `static` global array within the main code module. The main module contains all the measurement functions for the data cache elements of the memory hierarchy. The array is configurable in size using named constants and currently consists of 256 rows, each being 128KB in size, for a total size of 32MB. This is substantially larger than any currently known data cache. Hereafter, we refer to this data structure simply as “the array.”

The strategy of AMP is to first determine the line size (or TLB page size) for each level of the memory hierarchy in turn. Once the line size has been determined, AMP knows how often misses will occur in further experiments that access uncached memory regions at that level. For example, if the L1 D-cache line size is 32 bytes, then accessing a 1024

byte region of memory that is *not* present in the L1 D-cache should produce 1024/32 = 32 L1 D-cache misses. The approach for other parameters, such as total cache size and associativity, is to perform certain accesses that attempt to thrash that cache. If thrashing occurs, AMP will get the maximum possible miss rate for that access sequence, which is once per cache line. Details are in the sections below.

6.3.1 Data Cache Line Size

The line size of a data cache is determined using the following algorithm.

```
1. Flush first few rows of the array from the cache
2. Read current value of performance counter for cache misses
3. Read the first row of the array
4. Read value of the performance counter again
5. Compute the miss rate
6. Set line size to the power of two that is nearest to the recip-
   rocal of the miss rate
```

The L1 data cache will be our initial example.

Step 1: First, the data cache must have the first few rows of the array flushed from it. If a machine-dependent module is able to execute a cache flush instruction, it does so. Otherwise, AMP reads the second half of the array (i.e. the final 16MB of the array, at its current size) several times (not just once, in case a non-LRU replacement scheme is used.) If the first few rows of the array were ever present in the data cache, they should now be evicted by the later rows. This will be true as long as 16MB exceeds the size of any level of cache.

Steps 2–5: Turn on the *L1 data cache miss* performance counter, read its initial value, then read the first row of the array. Read the performance counter again, subtract its initial value from the new value to compute the number of L1 data cache misses that occurred. The number of misses is divided into the number of integers in one row of the array to give the miss rate, expressed as the proportion of integers that caused misses.

Step 6: The final step is to determine which power of 2 is nearest to the reciprocal of the miss rate. This is the cache line size. (AMP assumes that all cache line sizes are a number of bytes that is a power of 2, which is true for all data caches with which we are familiar.) The formula to compute the line size from the miss rate is:

$$LineSize = sizeof(int) \times 2^{\left\lfloor \frac{\log \frac{1}{MissRate}}{\log 2} + 0.5 \right\rfloor}$$

where *log* refers to the natural logarithm, which is divided by *log 2* to produce the logarithm base 2 of the reciprocal of the miss rate. This value is then rounded by adding 0.5 and applying the floor function (truncation). Raising 2 to this power gives us the number of integers in the cache line. A final multiplication by the number of bytes per integer converts the units to bytes.

For other levels of cache, the appropriate performance counter is used, and the same algorithm will produce the L2 line size, Data TLB (DTLB) page size, etc. Reliability of line size and page size computations is discussed in Section 6.2.2.

One unfortunate hurdle encountered in the validation of this algorithm was the presence of hardware errata producing invalid numbers from the L1 D-cache read miss counter on Sun UltraSparc-I and UltraSparc-II systems ([Sun Microsystems 1997]). This counter produces more read misses than there actually were data reads performed. We detected these anomalous numbers, and investigation turned up the errata sheets for the CPUs. AMP works around this problem by using a change in the code, controlled by compilation directives for UltraSparc targets, that uses *writes* instead of reads in the accesses to the first array row. The write miss counter is verified to work properly on these systems, and line size can be determined as accurately as on other CPUs.

6.3.2 Data Cache Size

Determination of the size of a data cache uses steps similar to determining the line size:

```
1. SizeHypothesis = MIN_CACHE_SIZE
2. while (SizeHypothesis is <= MAX_CACHE_SIZE) do
3.   Read SizeHypothesis bytes at
     beginning of the array
4.   Read performance counter for cache misses
5.   Reread SizeHypothesis bytes from beginning of array
6.   Reread performance counter
7.   if one miss per cache line then
8.     Exit the loop
9.   else
10.    Double the SizeHypothesis
11.endwhile
```

The algorithm iterates over cache size hypotheses starting with a defined minimum cache size. This is a named constant in the code that is currently 1024 bytes. (For modern processors that have performance counters, no cache will be this small, and the processors of several CPU generations ago that had cache sizes as small as 1KB did not have performance counters and will not be subject to our measurements.)

Steps 3–6: The first 1KB of the array is read to place it into the cache (if it will fit.) The appropriate performance counter is started, the hypothesized cache size is read a second

time, and the new value of the counter is read. A miss rate for this second pass through the hypothesized cache size is computed from the difference in the counter values read.

AMP defines the *expected failure miss rate* as the miss rate that it will see if the hypothesized cache size exceeds the actual cache size and the first read pass wrapped around the cache and evicted itself, causing the second read pass to miss once for each cache line. Thus, the expected failure miss rate is the reciprocal of the already-computed cache line size.

Steps 7–10: If the miss rate does not exceed a threshold fraction of the expected failure miss rate (a named constant, currently 0.80, empirically derived from experiments on several systems), then AMP concludes that the test data fit into the cache. In this case, the hypothesized size is doubled, and AMP iterates again.

When AMP finds a miss rate that indicates failure, it could assume that the *previous* hypothesis (the largest size that did *not* fail) is the cache size. However, not all cache sizes are powers of two; for example, the Alpha 21164 CPU has an on-chip unified secondary cache that is 96KB in size ([Compaq Computer Corporation 1998]). To accurately measure the size of such a cache, AMP iterates between the last non-failure test size and the first failed test size, in increments of 25% of the difference between them.

6.3.3 Data Cache Associativity

Given the cache size, the associativity can be found by experiments that try to thrash the cache.

```
1. AssocHypothesis = 1
2. while (AssocHypothesis < NbrOfCacheLines) do
3.  Access the array (2 * AssocHypothesis) times at spacings
   CacheSize / AssocHypothesis) bytes apart
4.  Read value of performance counter for cache misses
5.  Re-access the array (2 * AssocHypothesis) times at spacings
   (CacheSize / AssocHypothesis) bytes apart
6.  Re-read performance counter
7.  Compute miss rate
8.  if miss rate less than thrashing threshold then
9.    Increase AssocHypothesis to next
      feasible value
10.endwhile
11.if Thrashed then
12. return AssocHypothesis
```

```
13.else
14. return NbrOfCacheLines (i.e.
    fully associative)
```

Step 1: Start with direct-mapped as our hypothesis.
Step 2: The limit is a fully associative cache.
Steps 3-10: Choose an access pattern that would thrash if the true associativity matched our hypothesis. For example, if a primary data cache is known to be 16KB in size, then repeatedly and alternately accessing two array elements that are 16KB apart would create a miss rate close to 100% if the cache is direct-mapped (1-way associative), as the elements would map to the same cache line and evict each other from the cache. In this case, the function would terminate and return 1 as the associativity.
Steps 11-12: If thrashing is not observed, the hypothesis is advanced to the next integer that is a factor of the number of lines in the cache. For example, with a 16KB cache with 32 byte lines, there are 512 lines in the cache. The associativity should be a factor of 512. In this example, the factors are all powers of two, so the next associativity hypothesis is double the previous hypothesis, but this cannot be assumed. AMP will work correctly on machines with 3-way, 5-way, etc. caches.

In general, for an associativity hypothesis of *k*, AMP repeatedly and sequentially accesses *2k* array elements spaced *N/k* bytes apart, where *N* is the array size. For the 2-way associative hypothesis in the 16KB array, AMP accesses 4 elements at relative addresses within the array of 0, 8KB, 16KB, and 24KB. This will thrash a 2-way (but not a 4-way) associative cache. Testing proceeds until AMP sees cache thrashing.

6.3.4 Write Allocation Policy

AMP can determine whether a cache allocates a cache line upon a write miss. After flushing the cache, AMP writes to a region of the array that is smaller than the size of the cache being tested. Subsequent reads to the same region will be hits if the write misses caused cache line allocations, and will miss at the rate of once per line if the cache employs a no-write-allocate policy. Because AMP assumes that each downstream (larger) cache includes the entire contents of each upstream (smaller) cache, once AMP reaches a level in the cache hierarchy with a write-allocate policy, all downstream caches are assumed to be write-allocate caches, also.

6.3.5 Replacement Policy

The three common approaches to determining which set to replace after a cache miss are true LRU, pseudo-LRU, and random. The two LRU schemes will be similar for all

repeatable access sequences: a certain set will always be chosen for replacement for a given sequence of hits and misses. Depending on the pseudo-LRU implementation and the access sequence, true LRU and pseudo-LRU might choose different sets. Random replacement will choose a set to replace based upon some random value supplied by the system, and will not demonstrate repeatable behavior for all repetitions of an access sequence. AMP is thus able to distinguish between LRU and random. Ongoing work will differentiate pseudo and true LRU.

For 4-way and 8-way associative caches and TLBs, AMP accesses N array elements that map to the same set, where N is the associativity. Then it accesses the first $N-1$ elements again, followed by an $(N+1)$st element that maps to the same set. This last element will cause eviction of one of the first N elements. Turning on the appropriate cache miss counter and accessing element i will determine if element i was replaced. Repeating the entire experiment and iterating i from 0 to N-1 will give AMP a statistical picture of the replacement policy. If only a single set among the first N sets is ever replaced when the $(N+1)$st element is accessed, then some form of LRU replacement is being used. If the misses are scattered throughout all N sets, the replacement was random.

6.3.6 Data Cache Subblocking

If subblocking is employed in a data cache design, then a miss will cause only a portion of a block (i.e. line) to be fetched into the cache. The cache tags will be changed as if the entire line had been fetched, except that there are separate *valid bits* for each subblock in the line, and only the subblock that was fetched gets its valid bit set. For example, the vendor documentation for the UltraSparc-I and UltraSparc-IIi processors from Sun indicate that the L1 data cache has 32-byte blocks with 16-byte subblocks. If the use percentage within a cache line is poor, then time and bus cycles will be saved by not fetching 32 bytes on each cache miss, as it could be the case that one of the two subblocks would not be referenced before the whole line is evicted.

With respect to subblocking, there are two possibilities in the performance counter design. It could be that the L1 data cache miss counters record a miss on every subblock fetch, or they could record a miss only when a line is evicted and new tags are written. In the UltraSparc example, it could be that a miss is recorded every 16 bytes or every 32 bytes

of sequential access in uncached memory. It is desirable that AMP detect subblocking in both cases, which is a difficult requirement.

Detecting subblocking in the case where the performance counters record misses on every subblock fetch, e.g. every 16 bytes on the UltraSparc, is accomplished by noting that the difference between a data cache of a certain size and associativity with 16-byte lines, and a data cache of the same size and associativity with 16-byte subblocks, is that the former has more tags and hence more flexibility in mapping 16-byte chunks of memory into the cache. When the tags in a subblocked cache record that the first subblock is valid but the second subblock is not valid (or vice versa), there is only one 16-byte memory region that can be loaded into the empty subblock without evicting the valid subblock. Thus, if AMP reads from a memory address that is equal to the only subblock that can be loaded into the empty subblock, it will not evict the other subblock whether subblocking is employed or not. However, if AMP adds to that address the way size of the cache and reads from it, it will evict the first subblock if subblocking is employed, but it will not evict that subblock if it is in fact a complete block and subblocking is not employed. By alternating among accesses to subblocks that are at address k and address $k + linesize + waysize$, and choosing enough such pairs to fill up the associativity of the cache but no more, a subblocking cache will thrash while a cache without subblocking will have near-zero misses after the startup misses. This takes care of the case where the performance counters have caused AMP to report an n-byte line size, only to discover that it was an n-byte subblock size; AMP prints a corrective message before it finishes with the given level of cache.

The case in which the performance counters report a miss every n bytes, where n is the line size and $n/2$ is the subblock size, is more difficult. In that case, which is being left to the follow-up work for this research, AMP must use a combination of performance counters and CPU cycle counters to determine that the thrashing is occurring. If misses can be inferred from the significantly longer time for one access pattern compared to the other, then subblocking is employed. If the two access patterns take approximately the same time, then subblocking is not employed. In this case, the two addresses are k and $k + linesize/2 + waysize,$ because AMP is searching within the computed line size for subblocks.

6.3.7 Data Cache Miss Penalties

It has been shown that AMP has the ability to set up experiments that will either produce a near-zero number of cache misses or a large number that can be used to determine some cache parameter. As a result, it is quite easy for AMP to create an experiment that will produce a given number of data cache misses, once the line size, subblocking scheme (if any), total size, and associativity of the data cache have been determined. For example, once a certain region of the array has been evicted from the cache, AMP can repeatedly access the first cache line of the array, a total of *n* times, where *n* is usually set at 32 (to be explained below). This should produce a single compulsory miss followed by cache hits. If AMP instead accessed the first line *n-1* times and then accessed the second line once, then there would be two compulsory misses. In this manner, AMP can generate from 1 to *n* cache misses in experiments that each make *n* accesses to the array.

The key to determining miss penalties is to run experiments that each execute exactly the same number of instructions, make exactly the same number of memory accesses, differ only in the number of cache misses, and capture the time consumed by the CPU cycle counter. This counter is present in all machines with performance counters. With a sequence of experimental results in which n cache misses can be plotted against the number of CPU cycles consumed, varying n from 1 to 32, AMP can then plot a least-squares linear regression through the two-dimensional set of data points. The slope of this best fit line will be the data cache miss penalty in CPU cycles. The correlation coefficient will range from -1.0 to +1.0, with numbers close to +1.0 indicating a high enough correlation for the experiment to be trusted.

AMP accomplishes these experiments by using only the first row of the array and storing the column numbers of the array elements that need to be accessed in the experiment within that first row. The first column number used is at least one cache line beyond the end of the region used to store the column numbers. Using 32 values in the experiment gives AMP plenty of pairs of points for a valid least-squares best fit, while using only a small portion of the first row of the array to store columns. The setup of the experiment pro-

ceeds as follows:

```
for TargetMisses = 1 to 32 do
   store the column number of the first line to be accessed
      a total of (33-TargetMisses) times in the column storage area.
   store the next column number (next compulsory miss column) and
      increment column number, a total of (TargetMisses - 1) times
   access the array so as to flush the first row from the cache
   access the column storage region of the first row to get it
      back into the cache
   record initial cache miss counter and CPU cycle counter values
   for i = 0 to 31 do
      access array[0, array[0, i]]
   endfor
   record ending cache miss counter and CPU cycle counter values
endfor
```

The first store in the above pseudocode stores the number of times that the first cache line will be accessed, which causes one compulsory miss followed by hits. The second store makes the remainder of the accesses be to unique cache lines by striding through the array a cache line at a time, causing nothing but compulsory misses for the remainder of the experiment. The column storage sequence for the experiment is accessed to bring it into the cache just before starting the experimental access sequence, so that this regions does not cause additional cache misses. Thus, even though the inner loop access statement accesses the array twice, the inner access should always be a cache hit. The inner loop access loads the targeted array value into a variable declared as a `register` variable. AMP is compiled by the *gcc* compiler, which respects this keyword even without any optimizations being requested, so the variable into which AMP loads an array value does not cause any cache misses. The interface code to read the values of the performance counters often causes a small fixed number of cache misses, such as 1-3 misses. The only effect of these misses is to cause the cache miss versus time plot to start at 2-4 misses instead of 1.

Glossed over in the above pseudocode is the fact that AMP actually runs numerous repetitions of each experiment and takes the lowest CPU cycles and cache miss values from each experiment. These values are recorded in a vector whose index is cache misses and whose value is the lowest CPU cycles seen for that number of cache misses. Because AMP is seeking precision down to the CPU cycle, if possible, the repetition count can be varied

easily and has been set as high as 100 during testing. Values less than half that high are sufficient for any machine that is unloaded, i.e. does not have competing user processes.

When cache miss penalties are computed for the second level cache, the miss penalty will include the penalty for missing in the first level cache, which must occur before there can be a second level cache miss. When the data TLB miss penalty is computed, it will include the miss penalty cycles for any cache that is smaller in size than can be mapped by the TLB. This computation assumes that, if the TLB maps more memory than can fit into the L2 cache, then most TLB misses will happen only after L2 misses, whereas if the L2 cache holds more data than can be mapped into the data TLB, it can only be assumed that there is likely to be an L1 cache miss when a TLB miss occurs.

In practice, the experiments have proven extremely robust and accurate. Running in a single-user DOS environment, with no process competition, dozens of repetitions of an experiment are seen to produce exactly the same number of cache misses and CPU cycles. The correlation coefficients for many such machines have exceeded 0.99.

Note that, for the purposes of the data placement optimizations, all that is needed is a good estimate of the ratio of the L2 data cache miss penalty to the L1 data cache miss penalty, in order to know how to prioritize trade-offs in reducing conflicts in those two caches. An excellent estimate of the actual miss penalty in CPU cycles is useful to other users of AMP.

6.3.8 Data Cache: Split or Unified

It is essential to know whether a given level of cache is data-only or unified when analyzing performance or performing certain compiler optimizations. AMP can detect a unified cache by simply reading a portion of the array that equals the cache size, then executing a synthesized chunk of object code that is at least that size (but which does not perform data operations), and then re-read the portion of the array. If the second pass through the data array misses once per cache line, then the code execution must have evicted the data from the cache in question, and AMP concludes that it is unified.

An important result from the determination of unified or split status is that AMP can obtain some useful information in the absence of a complete set of performance counters. If a CPU has an L2 data cache miss counter, but not an L2 instruction cache miss counter,

AMP can characterize the line size, total size, associativity, and write policies of the L2 cache using the data counters only. After AMP determines that the cache is unified, the absence of an L2 instruction miss counter is not a problem. The same applies to unified TLBs.

6.3.9 Instruction Cache Line Size

Instruction cache line size is computed in the same way as the data counterpart, except that AMP executes a large block of straight-line code instead of accessing a region of a data array. The code block is generated from macros that repeatedly perform additions and subtractions on a pair of variables, leaving them with their initial values so that no overflow will occur when the macro is repeated thousands of times. The function containing the large block of code is compiled without optimization to ensure that the code does not disappear during compilation. The *gcc* compiler is used in order to take advantage of a non-standard extension to the C language that it provides, viz. the ability to take the object code address of a label in the code using the “`&&`” operator. This operator is used to compute the size of a block of code, which is then used (along with the cache miss counter values) to compute the miss rate. AMP then computes the line or page size directly from the miss rate using the equation from Section 6.3.1, dropping the `sizeof(int)` multiplication.

6.3.10 Instruction Cache Size

AMP computes the I-cache sizes using a function that contains a sequence of `switch` statements with 32 `cases` each, each `case` containing a code macro that generates 1KB of object code for the target machine. This macro is obtained from a header file that is generated automatically. A preliminary step in the software building process for AMP uses the *gcc* `'&&'` operator to compute object code sizes for the code macro, along with other pieces of code such as the surrounding control flow and a `NOP` instruction on the target machine. Using these sizes, the preliminary program generates a header file with macros that will expand to 1KB and 4KB of object code on the target machine, within a few bytes.

The function executes specified (via input parameters) `cases` within the `switch` statement to prime the instruction caches and TLB, then turns on the requested performance counter and executes the same `cases` again. As with the data case, AMP detects the

expected failure miss rate when it has exceeded the size of the cache, then performs a finer-grained search between the final pair of powers of two to get the final size.

As secondary and tertiary (L3) caches can be quite large, a clone of this function is provided in which each `case` of the `switch` statement has 4KB of object code from a macro instead of only 1KB. This function is called automatically as the size being tested exceeds the size that can be tested using 1KB macros.

In order to keep the size of these large blocks of synthetic code within very precise bounds, AMP measures, using the *gcc* `&&` operator, the code size produced by `if` and `switch` statements that surround the code macros. Every few `cases` within each `switch` statement, AMP uses a different macro that produces slightly less code in order to compensate for the overhead of this control code. Because *gcc* cannot compile a `switch` statement with 1024 `cases`, each of which has 1KB of code, AMP uses a sequence of 32 `switch` statements, each of which has only 32 `cases`, to achieve the code size needed. Even so, *gcc* can easily exhaust virtual memory limits on many systems. We also discovered an error in *gcc* code generation for such a large function on Compaq Alpha systems, which has been fixed by the *gcc* maintainers.

6.3.11 Instruction Cache Associativity

AMP computes associativity using the same function with the large `switch` statements, executing noncontiguous blocks of code located at relative spacings, just as it accessed array elements at certain relative spacings in Section 6.3.3.

6.3.12 TLB Measurements

All algorithms have been successfully tested and confirmed to produce valid results for instruction and data TLBs, where the analogous parameters are page size (instead of line size), TLB entries (instead of size in bytes), and associativity.

An interesting anomaly was detected on our MIPS R10000 CPU in an SGI Octane system running the IRIX operating system. Initial efforts to determine the number of TLB entries failed. Inspection of the raw data coming from the TLB miss counters showed that, as our SizeHypothesis neared the size reported in vendor documents (2 MB, in 64 entries that each map pairs of 16 KB sub-pages), thrashing occurred but then subsided, reducing

the miss rate to nearly zero. After much repetition and validation of these results, vendor employees confirmed that the IRIX OS can be configured to detect TLB thrashing and dynamically resize the pages to use more than 16KB per page. Thus, the repetitions of measurements, designed to increase robustness, gave IRIX time to detect repeated thrashing and eliminate it. AMP was redesigned to detect and report this dynamic resizing of pages, which has only been seen on IRIX systems to date.

Much to our surprise, AMP stopped reporting dynamic page resizing on a certain date. Our system administrators confirmed that a new release of IRIX had been installed, and they had not bothered to enable the page resizing. This confirmed the accuracy of AMP's page resizing detection algorithm.

6.4 Summary of Measurements

Table 6-1 summarizes the measurements collected for some popular machines.

System	L1I	L1D	L2	ITLB	DTLB
	Line/Size/Ass.	**Line/Size/Ass.**	**Line/Size/Ass.**	**Page/Size/Ass.**	**Page/Size/Ass.**
P5-133	32 / 8K / 2	32 / 8K / 2	N/C	4K / 32 / 2	4K / 64 / 4
PPro-200	32 / 8K / 2	32 / 8K / 2	32 / 256K / 4	4K / 32 / 4	N/C
PII-233	32 / 16K / 4	32 / 16K / 4	32 / 512K / 4	4K / 32 / 4	N/C
USparc-I	32 / 16K/ 2	32 / 16K / 1	64 / 512K / 1	N/C	N/C
R10K-225	64 / 32K / 2	32 / 32K / 2	128 / 1024K / 2	32K / 64 / 64	32K / 64 / 64

TABLE 6-1: Measured Cache and TLB Parameters

Associativity is measured in *number of degrees*; all other parameters are measured in *bytes*. N/C indicates that No Counter is available to count misses for that memory hierarchy element.

The systems used, respectively, are a 133MHz Pentium, 200MHz Pentium Pro, 233 MHz Pentium II, 167 MHz UltraSparc-I, and an SGI Octane with 225 MHz R10000 CPU. The Pentium CPUs have entirely different performance counter hardware (and software interfaces) than the Pentium-II/Pentium Pro family CPUs.

For TLB parameters, size is in *number of entries*. All machines had unified L2

caches; the R10000 has a unified TLB. All L1 caches were split. The P5 and UltraSparc L1 D-caches were no-write-allocate. No random replacement schemes were detected. The data cache subblocking of the UltraSparc machine could not be detected due to the previously mentioned hardware errata for the performance counters, and the other machines had no subblocking. Results were validated using the time consuming sources mentioned previously: vendor documentation, vendor personnel, etc.

The data miss penalties (in CPU cycles) for various Intel PCs, and the SGI MIPS-CPU system, are shown in Table 6-2. Again, hardware errata of the UltraSparc performance

System	L1 Data	L2 Data	DTLB
P5-133	13	N/C	37
PPro-200	4	37	N/C
PII-233	9	55	N/C
PII-333	19	73	N/C
R10K-225	4	65	75

TABLE 6-2: Miss Penalties for Various Intel CPU Personal Computers and SGI MIPS

counters precludes computing miss penalties. Deeper characterizations of code and data miss penalties is ongoing work. Note the increase in L2 miss penalties with increasing clock rate within the Pentium-II family. The decrease in L1 miss penalty from the Pentium (P5 CPU family) system to the other (P6 CPU family) systems is due to bringing the CPU and L1 caches together on a multichip module (MCM) in addition to other packaging and microarchitectural changes specifically designed to decrease the cache miss penalties. The ratios of L2 to L1 miss penalties are approximately 9, 6, 4, and 16 for the last four systems in the table, which have L2 miss counters. This work led to the initial default ratio of 8.0 being selected for the local refinement stage of the data placement optimizations.

6.5 Implementation

AMP is implemented in 12 modules of ANSI standard C comprising 29,214 lines of code including comments. It is compiled using *gcc* version 2.95.2. Run times range from 5 to 50 minutes on different systems (absence of certain counters speeds up the run times

considerably; TLB measurements are the most time consuming.) The executable file is approximately 9MB. Using a preprocessor symbol, this can be reduced to 800KB to create a DOS bootable diskette with a reduced version of AMP for Intel x86 PCs. The primary code reduction in this version is the removal of the majority of the synthetic code modules discussed in Section 6.3.10. This prevents AMP from determining the size of any L2 instruction cache that exceeds 256KB. As x86 PCs have a unified L2 cache, the L2 parameters will have already been determined using data cache measurements, and AMP will be able to determine that the L2 cache is unified if it does not exceed 512KB. The diskette version of AMP can be used to test multiple PCs quickly without installation of any APIs.

6.6 Software Availability

The file `ftp://ftp.cs.virginia.edu/pub/AMP/README` contains instructions for downloading executables for various target machines. Source code will be available here soon.

6.7 Ongoing Work

Enhancements nearing completion include instruction miss penalty characterization for instruction caches and ITLBs, timing-based detection of sub-block and sub-page schemes, and measurement of hardware elements such as branch target buffers. The miss penalty computations for various access patterns, such as a write miss that follows a read hit, a write miss that follows a read miss, read misses that are separated by n instructions that do not access memory (where n is varied from 0 up to the point at which the penalty curve flattens out), etc., are being designed and implemented. New target systems will be used as they become available, and measurements will be maintained at the FTP site.

6.8 Supplementing AMP with Microbenchmarks

Certain information needed for the compiler optimizations in this research cannot be obtained through performance counters. Instead, small synthetic microbenchmark programs have been written to answer key questions, primarily about machine dependent

timing concerns. The purpose of these synthetic benchmarks is to establish the relative benefits of various optimizations. These will be described briefly below.

6.8.1 Computing Benefits of Load and Store Coalescing

Computing the maximum possible benefit of coalescing loads can be done by timing an inner loop that executes millions of times, loading consecutive four-byte values from memory into two 32-bit registers. The assembly language output of the compiler can be modified by hand to load eight bytes at a time into a 64-bit register, then unpacking that register into a pair of 32-bit registers. Then the program is assembled and linked and retimed. For the UltraSparc-IIi systems timed in this research, coalescing 32-bit integer loads into a 64-bit integer load and a 64-bit shift instruction to unpack the upper 32 bits produced no time savings. As a result, no further effort was expended on this optimization.

The potential was greater for 32-bit floating-point load coalescing into 64-bit floating point loads, because each 64-bit floating-point register on the UltraSparc architecture is overlaid on a pair of 32-bit floating-point registers. Therefore, a 64-bit load into floating-point register `%d0` will have the effect of filling the 32-bit registers `%f0` and `%f1`. If the application program needs to load two 32-bit values from consecutive positions within a 64-bit-aligned memory address, the two 32-bit loads can be coalesced into a single 64-bit load, and no unpacking instruction is needed. The absence of the unpacking instruction means that two loads have been replaced with one load, with a net reduction in instructions executed (and also a reduction in code size), which should lead to a reduction in execution time.

However, timings on 500MHz UltraSparc-IIi machines showed no decrease in execution time. This is particularly surprising given the following recommendation from the UltraSparc-IIi User's manual, Chapter 21, "Code Generation Guidelines":

> UltraSparc-IIi supports single-cycle 8-byte data transfers into the floating-point register file for LDDF. Wherever possible, applications that use single-precision floating-point arithmetic heavily should organize their code and data to replace two LDFs with one LDDF. This reduces the load frequency by approximately one half, and cuts execution time considerably [Sun Microsystems 1997].

Unless the reduction in code space produces significant cache benefits, or the coalesced load reduces power consumption, the floating point load coalescing optimization

does not appear to be important for the UltraSparc-IIi machines.

The same experiment can be repeated using 64-bit floating-point stores to replace a pair of 32-bit floating-point stores. For the same 500 MHz UltraSparc-IIi machines, a 12% reduction in execution time was observed. Why the store coalescing provides a benefit, and not the load coalescing, cannot be explained from any available vendor documentation. For 225MHz and 300MHz SGI MIPS R10000 CPU systems, the reductions were 29% and 37%, respectively. Obviously, no real application could ever achieve such significant gains on these machines, but this kind of extremely tight loop will form the basis of the other synthetic benchmarks, which can then be compared for the magnitude of their reductions in execution time. Note especially the progressive increase in benefit as the CPU speed increases among the SGI systems.

When the integer experiment is repeated using store coalescing rather than loads, there is an increase in execution time. This is because it requires two CPU register-to-register instructions to pack a pair of 32-bit registers into a 64-bit register before the 64-bit coalesced store can be done, while it only required a single unpacking instruction after a 64-bit coalesced load. This is true of the UltraSparc and most desktop architectures, but not necessarily true for a wide variety of network processors, graphics processors, etc.

For the interaction with the instruction set architecture of particular machines, and the possibility of gaining more benefit from these optimizations on different architectures, see the discussion in Chapter 7. It should be noted here that the benefit of the optimization is based on the cost of a load instruction, which uses the CPU-to-cache bus, as compared to the cost of the unpacking instruction, which accesses only registers within the CPU. This benefit will increase as internal CPU speeds increase faster than CPU-to-cache bus speeds. Microarchitectural features such as load buffers and store buffers influence the timings. The sometimes surprising timing results, which can contradict vendor documents, underscore the importance of running these microbenchmarks before deciding which optimizations are worth implementing.

6.8.2 Computing Benefits of Immediate-Mode Addressing for Local Variables

Computing the benefit of substituting immediate-mode addressing for more expensive addressing with the local stack frame can be achieved by timing two inner loops. The first

accesses an array that falls within the range of the immediate offset from the frame pointer and uses the cheapest form of addressing. The second adds a dummy array that consumes the entire space that can be addressed with an immediate offset from the frame pointer, forcing the inner loop accesses to use the expensive addressing sequence.

For the pair of UltraSparc-I machines, the reduction in execution time was 56%. The same reduction was achieved for a 225 MHz SGI MIPS R10000 system. Again, no real application will realize this much benefit, but it does demonstrate that this optimization has more benefit per instance of its application than the load coalescing optimization.

6.8.3 Computing Benefits of Moving Cache Misses Earlier in Cache Lines

Given that many modern microarchitectures forward the requested item first when servicing a cache miss, as discussed in the context of cache line packing in Chapter 4, it is an interesting question whether there is any disadvantage in having the first miss in a cache line fall late in the line as opposed to early in the line. By timing three inner loops that cause millions of cache misses in the first word, last word, and a word near the middle of cache lines, it was determined that there was no measurable execution time difference on UltraSparc-I, UltraSparc-IIi systems, and MIPS R10000 systems.

6.8.4 Computing the Cost of Cache Thrashing

It is easy to code two loops such that one involves thrashing between two arrays that map to the same cache lines in memory, and the other loop has no such thrashing. For example, a dummy array can be placed in between the two arrays such that they collide in the cache in the thrashing version of the program. Removing the dummy array and eliminating the thrashing in the first-level cache will reduce execution time by slightly more than 40%.

When the arrays are made large enough to thrash the second level cache, rearranging the order of the arrays to eliminate thrashing reduces execution time by 52%. Both of these timings are for a 500MHz UltraSparc-IIi system.

6.9 Retargeting the Optimizations

There are several steps in retargeting the optimizations of this research to a new machine:

1. A performance counter API, such as PCL, should be installed on the new machine, and AMP compiled and linked on that machine. AMP can then be run in a few minutes to produce a characterization of the cache and TLB hierarchy of the machine. Note that AMP is already portable, as are PCL and other performance counter APIs upon which AMP depends, and AMP and these APIs do not need to be ported to the new machine. They only need to be compiled and linked on the new target machine.
2. The synthetic microbenchmarks that were used in Section 6.8 should be compiled and run on the new target machine to determine which optimizations are worthwhile.
3. The *vpo* header file *placement.h* for the new target should be updated to reflect the information gathered in step 1. The defined constants in this file for cache line size, associativity, etc., automatically drive the machine-dependent decisions in the data placement code for the new machine. The module consisting of files *placement.h* and *placement.c* should be copied from an existing target (e.g. the UltraSparc, currently), and modified to match the new target.
4. The front end that is used for the target should be updated to produce the metadata lines that pass information about actual parameters to *vpo*, if it is not a front end that has already been updated in this manner.
5. The code expander middleware, which translates the output of the front end into RTLs for the target machine, should be updated to pass the metadata lines through in the RTL file.
6. The machine-dependent code that gathers up the assembler lines for each global variable, and emits the assembler lines at the end of code generation, need to be cloned from an existing target (e.g. the UltraSparc) and updated for the new machine. This requires some understanding of assembler directives on the new machine.
7. The constant in *placement.h* that defines the immediate mode addressing range (e.g. 4KB for the UltraSparc) need to be updated for the new machine, if the range is small enough to justify the address range optimization for either locals or globals.
8. The few statements in *placement.c* that modify and replace RTLs to perform load coalescing should be modified to produce the correct RTLs for the new target machine. The detection of the RTLs to be replaced should not need to be modified. This step and the next step are only performed if step 2 determined that either load or store coalescing is beneficial.
9. The machine description for the new target machine should be augmented to include the 64-bit instructions needed to perform the coalescing operations in step 8 (e.g. the *ldx* and *srlx* instructions on the UltraSparc).
10. The machine-dependent code in function `fixentry()` that emits the prologue and epilogue code for each function needs to have RTLs added to keep all stack frames aligned on second-level cache line boundaries. This step is also only needed if at least one optimization of local variables is judged to be beneficial from the items discussed in steps 7 through 9.

The overall effort for making and testing these changes for a new target is probably in the range of 2-4 weeks. After the research has been retargeted to at least two more machines in addition to the current UltraSparc target, it will become obvious which sections of code were simply copied and did not need to be modified. This will be likely to include most of the *placement.c* code. This code can then be moved almost entirely into the machine-independent module that includes most of the data affinity analysis and data

placement code already, with a few small machine-dependent code fragments left behind in small functions to be called from the machine-independent code. At this point, well over 90% of the *vpo* code from this research will be in machine-independent modules.

Chapter 7

Implications for Hardware and Software Design

The optimizations performed in this research, and the measurements performed, give certain insights into issues in machine architecture, microarchitecture, and software engineering. These will be briefly discussed in this chapter.

7.1 Architecture Interactions

The microbenchmarks presented in Chapter 6 demonstrated no positive or negative change for the load coalescing optimization on the UltraSparc-IIi systems. This result held for both integer and floating-point variables. Store coalescing of 32-bit floats into 64-bit floats was beneficial, while the similar transformation for integers was harmful, due to the need for two instructions to pack the two 32-bit integer values into a 64-bit register before the store could be performed.

The problems with load and store coalescing of integers are caused by the need to use packing and unpacking instructions. One recent work in coalescing was able to achieve execution time reductions on an Intel StrongARM IXP-1200 network processor [Nandivada and Palsberg 2003]. This processor permits a 64-bit load to specify two consecutive 32-bit registers as the destinations. If the two values can be targeted to those two registers by the register allocation and register assignment procedures in the compiler, then there is no need for any unpacking instructions at all. The same is true for storing two consecutive registers to an 8-byte memory location.

Comparison of the various load and store coalescing operations on the UltraSparc-IIi and the StrongARM IXP-1200 show that the presence of loads with two consecutive registers as the destinations, and of stores with two consecutive registers as the sources, avoids any need for packing and unpacking instructions. This makes the coalescing optimization beneficial rather than neutral or even harmful. These loads are implemented in a RISC (reduced instruction set computer) architecture in the case of the StrongARM CPU core,

and should be feasible for many other architectures. In fact, a recent trend instruction sets is to provide additional instructions and registers for multimedia processing, with loads and stores of various widths, at least up to the CPU-to-cache bus width [Larsen and Amarasinghe 2000]. The only reason such load and store instructions are not available for the primary register sets is that these architectural extensions were designed years after the CPU was designed. Future designs can avoid this lack of orthogonality.

No special load instructions are needed if each 64-bit register is overlaid on a pair of 32-bit registers, as in the UltraSparc floating-point register set. In the UltraSparc integer registers, however, each 64-bit register contains a 32-bit addressable register in its lower 32 bits, while its upper 32 bits are not a separately addressable register. This creates the need for packing and unpacking instructions, whereas the floating-point register set just needs register targeting in the compiler's register assignment implementation in order to assign adjacent 32-bit registers to variables that are involved in load or store coalescing.

The previous discussions of addressing mode optimizations based on the range of values available in immediate offset addressing mode, suggest that computer architects can assist in this optimization by allocating more bits to the immediate offset field in the load and store instructions. The feasibility of this allocation depends on numerous other decisions made when designing the instruction set, of course. The MIPS R10000 and related CPUs are an example of instruction sets with 16 bits for immediate offsets, rather than the 13 bits of the UltraSparc family [MIPS Technologies 1996]. This means that optimization of global variable addressing can occur when two or more globals have base addresses within 32KB distance of each other, rather than within 4KB as on the UltraSparc machines. It also means that a stack frame would have to exceed 32KB in size, rather than 4KB, before any expensive addressing of locals occurs. Very few stack frames are that large. On such a machine, the stack frame addressing optimization for locals would hardly be worth implementing, while even greater opportunities would be provided for reducing the addressing costs of globals.

Controlling the alignment of all arrays on cache line boundaries makes it easier to analyze the alignment of loads and stores, and to analyze at what point a sequence of loads (e.g. within an unrolled loop) crosses a cache line boundary and generates a potential compulsory cache miss. In recent work by Larsen et al., it was demonstrated that if the compiler

could guarantee that a certain load is a cache hit in the first-level cache (because it follows another load from the same cache line), then it could issue a special load that does not check the cache tags to confirm that it is a hit [Witchel et al. 2001], [Larsen et al. 2002]. With this addition to the instruction cache architecture and the support from the compiler, up to a 35% reduction in data cache power consumption could be realized. With a data placement framework that controls the addresses and alignment of all global variables, with all arrays aligned on second-level cache line boundaries, the potential for such a hardware/software synergy is maximized, and the analyses required are simplified. Determining the address of each loaded data item is the first step towards realizing these power savings. (The second step is convincing the architects that they should trust compiler writers with a load that bypasses the cache tag checks.)

7.2 Implications for Microarchitecture Implementation

A number of microarchitectural details that relate to the cache and memory hierarchy are apparently based on the assumption that the machine will be called upon to execute programs that were not compiled with data placement optimizations or other efforts to improve locality and reduce cache misses. If the designer of an embedded system CPU knew that the system would be used with the optimizing compiler supplied by his company, and he knew that this compiler performed the data placement optimizations of this research, then it is possible that some of the microarchitectural resources that have often been expended on certain cache and TLB issues could be eliminated. Embedded systems are mentioned in particular because of the concern for reducing hardware complexity (and hence cost, size, power consumption, etc.) in such systems, and also because of the greater possibility that a particular embedded processor will be used only by customers who will be using the optimizing compiler and will be very conscious of code efficiency.

One example of potential reduction of hardware complexity comes from the discussion of wrap-around cache line fill hardware in previous chapters. With all arrays aligned on cache line boundaries, and scalars packed into cache lines with an ordering criterion that places the first-accessed variable before others (on average), the distribution within the cache line of items causing cache misses should be improved on all codes that are not mak-

ing odd strides through arrays.

It should also be possible to reduce some complexity in coalescing load and store buffers, if load and store coalescing are beneficial and are performed by the compiler. Combining loop unrolling and access coalescing of array elements with access coalescing of scalar variables should greatly reduce the need for hardware coalescing within load or store buffers. In fact, the presence of such hardware helps to reduce the benefit of the compiler optimization, creating a chicken-and-egg problem. A simple, low-power embedded system could be better off with compiler-based coalescing.

Discussion of the speedup achieved by the program *compact* in Chapter 5 mentioned the performance issues associated with having multiple cache misses outstanding at once. Although the simulation tools used in this research cannot measure this phenomenon, it could be that microarchitectures will have less pressure on the affected resources in the presence of a data placement optimization.

Another example of microarchitecture design assuming the worst about cache misses is the use of subblocking in caches and subpaging in TLBs. Such schemes make sense primarily when the usage level within a cache line (or page) is well below 100%, so that there is a high probability that fetching more bytes would merely be pollution of the cache line (or page). With increased locality throughout the data space of the program, including within the packed cache lines, the benefits of subblocking might not be worth the implementation cost, including the extra valid bits in the cache. In addition, subblocking gives up potential spatial locality benefits by forcing two fetch operations on every cache line that is used to a high degree before eviction.

Random replacement schemes in caches are another example of addressing pathological worst cases under the assumption that the stream of cache misses is a chaotic and unmanaged sequence. In fact, when a data placement optimization has significantly increased locality throughout the memory hierarchy, and removed the worst instances of cache thrashing from the program, a random replacement scheme will suffer more costs from replacement misses than it will accrue benefits from decreased thrashing. Of course, associativity increases are less beneficial in the presence of good data placement.

While the author is biased in favor of putting work into compilers and reducing corresponding hardware complexity, there is one aspect of recent complex microarchitectures

that matches this research quite well. Many CPUs employ a form of *speculative execution*, in which the processor executes beyond a branch instruction before it is able to determine with 100% accuracy the direction that will be taken by the branch [Smith et al. 1990]. If the branch prediction was inaccurate, then some executed instructions will have their results discarded. If some of these instructions were loads, then unnecessary activity was created in the cache and memory hierarchy. Because this activity is not apparent to a dynamic profiling run through the program, which will see the branch direction that is actually taken, not all of the cache activity is apparent through the dynamic profile. Static data affinity analysis, however, follows all paths that are within loops and have significant execution frequency estimates. This causes temporal affinity between a variable accessed just before a branch and another variable accessed just after the branch to be recorded, regardless of which branch direction is taken on a particular run of the program (although there may be different probabilities assigned to the two directions). It can be seen that speculative execution does not have the same negative interaction with static data affinity analysis as it would have with dynamic profiling techniques. (However, the author does not want to be understood to be encouraging this kind of extra memory activity.)

7.3 Implications for Software Engineering Practices

There are several interactions of this research with software practices. First, the software engineer should be free to declare variables in the order that makes the code most readable, with the knowledge that the data placement optimization will take care of cache concerns. It could be more readable to declare an array and then immediately declare the scalars that are used as indices, counters, sums of array elements, etc. The fact that only two or three scalars might be declared between two arrays, which might ordinarily cause padding bytes to be inserted, or cause the second array to start in the middle of a cache line, will no longer be a concern of the software engineer.

Many software engineers declare variables with a greater width than is actually required by their use in the program. This is encouraged by the fact that certain operations in the typical RISC machine take the same amount of time regardless of width. For example, it is common for a 32-bit floating-point addition to take the same number of CPU

cycles as a 64-bit floating-point addition [DeLano et al. 1992]. Under these circumstances, it seems prudent to avoid potential rounding, overflow, and precision problems by using the wider data type in all declarations.

This research points out several problems with using wider data types than is necessary. First, no load or store coalescing is possible on most machines with the widest data types. A second problem is that address range optimizations will be possible less often if various arrays are twice as large as they need to be, as it decreases the probability of the starting addresses of two globals being within a certain range of each other. Perhaps most importantly, doubling the size of an array could move it from one data size category to another, as defined in Table 5-2. For example, an array could become larger than the size of one of the levels of cache if it is declared with twice the actually required precision. It will certainly make it harder to position arrays with swaps and rotations during local refinement of the data placement so that conflicts are avoided if the arrays double in size.

Informal communication indicates that many scientific programs absolutely must have the benefits of double precision floating point variables rather than single precision, but applications in gaming, multimedia, and signal processing can often use single-precision arithmetic without problems. Even among the scientific programs in the Perfect Club benchmark suite, *ADM*, *DYFESM*, *FLO52*, and *SPEC77* primarily use single-precision data and arithmetic.

Chapter 8

Conclusions and Future Work

8.1 Summary of Contributions of the Research

It is best to look at the entire scope of the multi-goal data placement framework in order to fully understand the contributions of this research. Prior work focused on reducing first-level cache misses and used lengthy dynamic profiling runs that would not be tolerated by most potential users of the optimizations. Extensions to multilevel and associative caches were difficult to conceive and not ever implemented. Page locality was not a design goal and was sometimes harmed as a result. Bottom-up greedy algorithms were employed to place data items one at a time. Addressing optimizations and access coalescing have been implemented in isolation from a comprehensive data placement optimization.

In contrast, the present research creates a data placement framework that addresses many goals at once:

- **Reducing the weighted sum of data cache misses throughout a multilevel cache hierarchy.**
- **Improving page locality.**
- **Improving locality within cache lines.**
- **Moving compulsory cache misses closer to the beginning of each cache line.**
- **Increasing the opportunities for memory access coalescing.**
- **Increasing the opportunities for addressing range optimizations.**
- **Laying the foundation for analysis of code/data conflicts in unified levels of the cache hierarchy.**

All of these goals were pursued within the constraints of static analysis in order to make the optimizations usable in practice.

Despite the disadvantages of pursuing multiple goals at once and using only static

analysis, the data placement optimization achieved a greater reduction in first-level data cache misses than prior work that targeted nothing else besides that goal. This raises several (non-exclusive) possibilities in the final evaluation of this research:

- **It could be that static data affinity analysis does not lag as far behind dynamic profiling methods as prior research would indicate.**
- **The global, top-down approach to data placement via hierarchical graph partitioning, when combined with a powerful local refinement algorithm, could be superior to prior data placement algorithms.**
- **The design of the multi-goal data placement framework is able to coherently address multiple goals without being forced to make trade-offs that would compromise its effectiveness in reducing cache misses.**

A case can be made for each of these possibilities based on the research that has been explained and presented in this dissertation. The static data affinity analysis is an original contribution in its own right, and could be quite competitive with dynamic profiling methods. The design of the data placement optimizations, by integrating a top-down hierarchical graph partitioning approach with a refinement algorithm based on a cache map and cache conflict metric, is able to address multiple cache levels and any degree of associativity as a natural part of the design. The difficulties encountered by prior research with respect to these two aspects of real cache hierarchies simply never arise as difficulties at all in this research. By fine-tuning the use of the *Chaco* graph partitioning software, the locality required for both addressing range optimizations and TLB/paging improvement is created without compromising the goal of reducing cache misses. Finally, by maintaining access order information throughout the static data analysis stage, the cache line packing optimization is able to maximize opportunities for access coalescing while also minimizing cache line pollution and moving compulsory cache misses closer to the beginning of each cache line in order to facilitate the simplification of hardware.

With careful design, the multiple goals of this data placement framework are cooperative rather than competing goals.

The major contributions of the research can be concisely summarized:

- **A static data affinity analysis infrastructure that incorporates numerous original ideas and gathers the information needed to improve multiple performance goals.**
- **A performance-counter based program to characterize numerous aspects of the cache and TLB hierarchy of any machine with performance counters, to make use of this research practical, as well as satisfying the characterization needs of various other researchers in other fields.**
- **A multi-goal data placement optimization framework that addresses numerous performance goals in an intelligent manner without compromises or trade-offs.**
- **Insights into the relationship of various optimizations to each other and to data placement, and into the relationship between data placement and various design decisions in computer architecture and software engineering.**

8.2 Future Enhancements and Extensions of the Research

The lengthy process of implementing the multi-goal data placement framework and measuring its effects motivates certain desired extensions and enhancements of the work. These are listed briefly below, roughly prioritized according to the intended order of addressing these issues.

1. An ongoing search for suitable benchmarks and input data sets must continue.
2. Incorporate prior work in static branch prediction and measure the effect on the accuracy of the static data affinity analysis. Enhance the determination within *vpo* of the number of iterations of each loop using interprocedural analysis, also to increase the accuracy of the static analysis.
3. Complete implementations of access coalescing and address range optimizations.
4. Find machines that are more likely to benefit from access coalescing and port to them.
5. Experiment with additional means of covering the local refinement search space to increase the confidence that the optimization is not getting stuck in a local optimum.
6. Integrate array subscript analysis into the analysis and placement.
7. Place the code segment so as to minimize code versus data conflicts in the unified cache.

These enhancements should continue to confirm the soundness and usefulness of the multi-goal data placement framework design in its current form, while yielding even greater performance improvements on a variety of benchmark programs.

References

ABU-SUFAH, W. 1979. Improving the performance of virtual memory computers. Ph.D. thesis, University of Illinois at Urbana-Champaign.

AMMONS, G. AND LARUS, J. R. 1998. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 72–84.

ANNAVARAM, M., PATEL, J. M., AND DAVIDSON, E. S. 2001. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*. ACM Press, New York, New York, United States, 52–61.

AUSTIN, T. 1996. Hardware and software mechanisms for reducing load latency. Ph.D. thesis, University of Wisconsin at Madison.

BACON, D. F., CHOW, J.-H., CHING R. JU, D., MUTHUKUMAR, K., AND SARKAR, V. 1994. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*. Integrated Solutions, Toronto, Ontario, Canada, 270–282.

BALL, T. AND LARUS, J. R. 1993. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 300–313.

BALL, T. AND LARUS, J. R. 1996. Efficient path analysis. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, San Mateo, California, United States, 46–57.

BENITEZ, M. E. 1994. Register allocation and phase interactions in retargetable optimizing compilers. Ph.D. thesis, University of Virginia.

BERRENDORF, R. AND ZIEGLER, H. 2003. PCL: The performance counter library, version 2.2.

BODIN, F. AND SEZNEC, A. 1995. Skewed associativity enhances performance predictability. In *Proceedings of the 22nd annual international symposium on Computer architecture*. ACM Press, New York, New York, United States, 265–274.

BOEHM, H.-J. 2000. Reducing garbage collector cache misses. In *Proceedings of the second international symposium on Memory management*. ACM Press, New York, New York, United States, 59–64.

BOLAND, L., GRANITO, G., MARCOTTE, A., MESSINA, B., AND SMITH, J. 1967. The IBM System/360 model 91: Storage system. *IBM Journal of Research and Development 11,* 1 (Jan), 54–68.

BROWN, A. B. AND SELTZER, M. I. 1997. Operating system benchmarking in the wake of lmbench: a case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, New York, New York, United States, 214–224.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News 25,* 3, 13–25.

CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. 1998. Cache-conscious data placement. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, New York, United States, 139–149.

CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. 1994. Compiler optimizations for improving data locality. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, New York, United States, 252–262.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 286–297.

CHEN, T.-F. AND BAER, J.-L. 1992. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, New York, United States, 51–61.

CHILIMBI, T. M. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 191–202.

CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. 1999a. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 13–24.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999b. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 1–12.

CHILIMBI, T. M. AND LARUS, J. R. 1998. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the first international symposium on Memory management*. ACM Press, New York, New York, United States, 37–48.

CMELIK, B. AND KEPPEL, D. 1994. Shade: a fast instruction-set simulator for execution analysis. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. ACM Press, New York, New York, United States, 128–137.

COLEMAN, C. L. AND DAVIDSON, J. W. 2001. Automatic memory hierarchy characterization. In *Proceedings of the IEEE 2001 International Symposium on Performance Analysis of Systems and Software*. IEEE Press, San Mateo, California, United States, 103–110.

Compaq Computer Corporation 1998. *Alpha Architecture Handbook, Version 4*. Compaq Computer Corporation.

COPPERSMITH, D. AND RAGHAVAN, P. 1989. Multidimensional on-line bin packing: Algorithms and worst-case analysis. *Operations Research Letters 8,* 1 (Feb), 17–20.

CYBENKO, G., KIPP, L., POINTER, L., AND KUCK, D. 1990. Supercomputer performance evaluation and the perfect benchmarks. In *Proceedings of the 4th international conference on Supercomputing*. ACM Press, New York, New York, United States, 254–266.

DAVIDSON, J. W. AND FRASER, C. W. 1980. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst. 2,* 2, 191–202.

DAVIDSON, J. W. AND JINTURKAR, S. 1994. Memory access coalescing: a technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 186–195.

DAVIDSON, J. W. AND WHALLEY, D. B. 1991. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems 15,* 9, 459–472.

DELANO, E., WALKER, W., YETTER, J., AND FORSYTH, M. 1992. A high speed superscalar PA-RISC processor. In *Spring COMPCON '92 Digest of Papers*. IEEE Press, San Mateo, California, United States, 116–121.

DELORIE, D. 2001. DJGPP: A free 32-bit development system for DOS, version 2.03.

DUNLOP, A. AND KERNIGHAN, B. W. 1985. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 4,* 1 (Jan), 92–98.

FENLASON, J. AND STALLMAN, R. 1992. *GNU gprof: The GNU profiler.* Free Software Foundation, Inc., Cambridge, Massachusetts, United States.

GABOW, H. N. 1973. Implementations of algorithms for maximum matching on nonbipartite graphs. Ph.D. thesis, Stanford University.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, New York, United States.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. 21,* 4, 703–746.

GLOY, N., BLACKWELL, T., SMITH, M. D., AND CALDER, B. 1997. Procedure placement using temporal ordering information. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, San Mateo, California, United States, 303–313.

GRUNWALD, D., ZORN, B., AND HENDERSON, R. 1993. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 177–186.

HABER, G., KLAUSNER, M., EISENBERG, V., MENDELSON, B., AND GUREVICH, M. 2003. Optimization opportunities created by global data reordering. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2003)*. IEEE Computer Society Press, San Mateo, California, United States, 228–239.

HARA, T., ANDO, H., NAKANISHI, C., AND NAKAYA, M. 1996. Performance comparison of ILP machines with cycle time evaluation. In *Proceedings of the 23rd annual international symposium on Computer architecture*. ACM Press, New York, New York, United States, 213–224.

HASHEMI, A. H., KAELI, D. R., AND CALDER, B. 1997. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 171–182.

HELLER, D. 1999. Rabbit: A performance counters library for Intel/AMD processors and linux.

HENDRICKSON, B. AND LELAND, R. 1995a. *The Chaco User's Guide: Version 2.0*. Sandia National Laboratories, Albuquerque, New Mexico, United States.

HENDRICKSON, B. AND LELAND, R. 1995b. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing 16,* 2, 452–469.

HENDRICKSON, B. AND LELAND, R. W. 1995c. A multi-level algorithm for partitioning graphs. In *Supercomputing*. ACM Press, New York, New York, United States.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture: A Quantitative Approach*, First ed. Morgan Kaufmann Publishers, San Mateo, California, United States.

Intel Corporation 1997. *Intel Architecture Software Developer's Manual: Volume 2: Instruction Set Reference*. Intel Corporation.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, New York, United States, 319–329.

JOHNSON, T. L. 1998. Run-time adaptive cache management. Ph.D. thesis, University of Illinois at Urbana-Champaign.

JOUPPI, N. P. AND WILTON, S. J. E. 1994. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21ST annual international symposium on Computer architecture*. IEEE Computer Society Press, San Mateo, California, United States, 34–45.

JOURDAN, S., SAINRAT, P., AND LITAIZE, D. 1995. Exploring configurations of functional units in an out-of-order superscalar processor. In *Proceedings of the 22nd annual international symposium on Computer architecture*. ACM Press, New York, New York, United States, 117–125.

KALAMATIANOS, J. 2000. Microarchitectural and compile-time optimizations for performance improvement of procedural and object-oriented languages. Ph.D. thesis, Northeastern University.

KANDEMIR, M. T. 2001. A compiler technique for improving whole-program locality. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, New York, United States, 179–192.

KERNIGHAN, B. W. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal 49,* 2, 291–307.

KISTLER, T. AND FRANZ, M. 2000. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst. 22,* 3, 490–505.

LARSEN, S. AND AMARASINGHE, S. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 145–156.

LARSEN, S., WITCHEL, E., AND AMARASINGHE, S. 2002. Increasing and detecting memory address congruence. In *Proceedings of the 11th international conference on Parallel Architectures and Compilation Techniques (PACT 2002)*. IEEE Computer Society, San Mateo, California, United States, 18–29.

LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 259–269.

LUK, C.-K. AND MOWRY, T. C. 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, New York, United States, 222–233.

LUK, C.-K., MUTH, R., PATIL, H., COHN, R., AND LOWNEY, G. 2004. Ispike: A post-link optimizer for the Intel Itanium architecture. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)*. IEEE Computer Society Press, San Mateo, California, United States.

MCFARLING, S. 1989. Program optimization for instruction caches. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, New York, United States, 183–191.

MCKEE, S. A., ALUWIHARE, A., CLARK, B. H., KLENKE, R. H., LANDON, T. C., OLIVER, C. W., SALINAS, M. H., SZYMKOWIAK, A. E., WRIGHT, K. L., WULF, W. A., AND AYLOR, J. H. 1996. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 10th international conference on Supercomputing*. ACM Press, New York, New York, United States, 125–132.

MCVOY, L. AND STAELIN, C. 1996. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Usenix Technical Conference*. USENIX Association, Berkeley, California, United States, 279–295.

MICHALEWICZ, Z. AND FOGEL, D. B. 2000. *How to Solve It: Modern Heuristics*. Springer-Verlag, Heidelberg, Germany.

MIPS Technologies 1996. *MIPS R10000 Microprocessor User's Manual, Version 2.0*. MIPS Technologies.

MORANO, D., KHALAFI, A., KAELI, D. R., AND UHT, A. K. 2003. Realizing high IPC through a scalable memory-latency tolerant multipath microarchitecture. *SIGARCH Comput. Archit. News 31,* 1, 16–25.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, New York, United States, 62–73.

MUCCI, P., BROWNE, S., DEANE, C., AND HO, G. 1999. PAPI: A portable interface to hardware performance counters. In *Proceeedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference*. Department of Defense, Washington, DC, United States.

NANDIVADA, V. K. AND PALSBERG, J. 2003. Efficient spill code for SDRAM. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*. ACM Press, New York, New York, United States, 24–31.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1997. Memory data organization for improved cache performance in embedded processor applications. *ACM Trans. Des. Autom. Electron. Syst. 2,* 4, 384–409.

PAPADIMITRIOU, C. H. AND STEIGLITZ, K. 1998. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, Mineola, New York, United States.

PATTERSON, J. R. C. 1995. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 67–78.

PETRANK, E. AND RAWITZ, D. 2002. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, New York, United States, 101–112.

PETTIS, K. AND HANSEN, R. C. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 16–27.

PUGH, W. 1994. Counting solutions to presburger formulas: how and why. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 121–134.

Pure Software 1995. *Quantify User's Guide*. Pure Software, Sunnyvale, California, United States.

SEIDEN, S. S. AND VAN STEE, R. 2002. New bounds for multi-dimensional packing. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, United States, 486–495.

SHUF, Y., GUPTA, M., BORDAWEKAR, R., AND SINGH, J. P. 2002. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, New York, United States, 295–306.

SMITH, M. D., LAM, M. S., AND HOROWITZ, M. A. 1990. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th annual international symposium on Computer Architecture*. ACM Press, New York, New York, United States, 344–354.

SOLIHIN, Y., LEE, J., AND TORRELLAS, J. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, San Mateo, California, United States, 171–182.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 196–205.

SRIVASTAVA, A. AND WALL, D. W. 1994. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 49–60.

Standard Performance Evaluation Corporation 2002. *SPEC CPU2000 V1.2 Documentation*. Standard Performance Evaluation Corporation.

Sun Microsystems 1997. *UltraSparc-IIi User's Manual*. Sun Microsystems.

WAGNER, T. A., MAVERICK, V., GRAHAM, S. L., AND HARRISON, M. A. 1994. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 85–96.

WEAVER, D. L. AND GERMOND, T. 2000. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, Englewood Cliffs, New Jersey, United States.

WILLIAMS, R. 1991. Performance of load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience 3*, 457–481.

WILSON, K. M. AND OLUKOTUN, K. 1997. Designing high bandwidth on-chip caches. In *Proceedings of the 24th annual international symposium on Computer architecture*. ACM Press, New York, New York, United States, 121–132.

WITCHEL, E., LARSEN, S., ANANIAN, C. S., AND ASANOVIC, K. 2001. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, San Mateo, California, United States, 124–133.

WOLF, M. E. AND LAM, M. S. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. ACM Press, New York, New York, United States, 30–44.

ZHAO, W., CAI, B., WHALLEY, D., BAILEY, M. W., VAN ENGELEN, R., YUAN, X., HISER, J. D., DAVIDSON, J. W., GALLIVAN, K., AND JONES, D. L. 2002. VISTA: a system for interactive code improvement. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*. ACM Press, New York, New York, United States, 155–164.