ANALYZING AND TUNING MEMORY PERFORMANCE IN SEQUENTIAL AND PARALLEL PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Margaret Rose Martonosi
January 1994

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Anoop Gupta
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Thomas Anderson

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Albert Macovski

Approved for the University Committee
on Graduate Studies:

Acknowledgments

In this section I gratefully acknowledge some of the people who offered me help throughout graduate school.

First, I thank my advisor, Professor Anoop Gupta, whose work ethic and passion for perfection offer such a strong example for all of his graduate students. His many insights and comments on the research, papers, and talks that led to this dissertation have been important to me as I made the transition from student to researcher. I am also grateful to my associate advisor, Professor Thomas Anderson, for his input and encouragement along the way. Professor Monica Lam offered useful suggestions for the work as a member of my thesis orals committee. Thanks also to Professor Albert Macovski for serving as my third reader and for chairing my orals committee.

I gratefully acknowledge the financial support given me by a fellowship from the National Science Foundation and research funding from the Advanced Research Projects Agency.

Members of the Stanford DASH group made contributions which helped this work as well. Dr. Stephen Goldschmidt developed the Tango Lite system which has been so useful in building MemSpy, as well as in so much other Stanford research. Dr. Edward Rothberg, Maneesh Agrawala, Phil Lacroute and others contributed applications which formed MemSpy case studies.

My good friends Dr. Paul Barbone, Brendan Del Favero, and Chris Pohalski spiced up life here at Stanford, and Brendan deserves special mention for proofreading a draft of this work in the final days. Thanks also to fellow CIS denizens Don Ramsey and Phil Lacroute, for the fun ski trips and crossword puzzle lunches, and also to Tony Todesco, for being such a considerate officemate and kind friend. My housemates at 167 Hillside have been great as well — particularly Jeff Welser, who had the decency to graduate after me! Finally, I especially want to acknowledge two fellow EE women: Dr. Susan Lord and Kathy Richardson, for all their friendship and understanding along the way.

My family has also been amazingly supportive of me all along. Thanks to my sister Susan and my brother Anthony for the amusing phone conversations and glimpses of life outside academia, and also to my aunt, Leonore Gouvea, who has always been such an enthusiastic cheerleader. I am especially grateful to my sister Mary Anne and her husband Dan. They made me feel welcome to stop by their house in Redwood City anytime, provided me with lots of fun distractions (the dearest of which is my two year old nephew Stephen!), and dished up a lot of free meals.

Most of all, I thank my parents, Anthony and Mary Martonosi, who always placed such a high priority on the growth and education of their four children. As I complete this dissertation, I dedicate it to them with more love and gratitude than I can express here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Processor speeds in modern computers are improving at a much faster rate than main memory speeds, and as a result, relative latencies from processors to main memory have increased. In uniprocessors, main memory latencies can be tens of processor cycles, while in multiprocessors, latencies can be over a hundred cycles. These figures are likely to be even larger in the future. Architects have responded to this trend by introducing memory hierarchies, where one or more levels of cache are imposed between the processor and main memory. However even with these hierarchies, many programs still have poor performance due to memory stalls.

To improve program memory performance, compilers and programmers can often transform the application so that its memory referencing behavior takes better advantage of the memory hierarchy. The challenge in performing these transformations, however, is that an application's referencing behavior and interactions with the memory system are often difficult to statically analyze or reason about. The high-level information collected by many existing performance monitoring tools is often not sufficiently detailed to analyze specific memory performance bugs. Thus, the focus of this work is on devising techniques to efficiently collect detailed statistics on application memory behavior and to effectively organize the large volumes of data collected. This information can then be used to guide programmers or compilers towards the program transformations that will be most effective in improving program performance.

1.1 Problem Statement

Ultimately, the motivation for this research stems from trends in processor and main memory speeds. For more than a decade, improvements in processor cycle times have outpaced improvements in Dynamic RAM (DRAM) speeds [HP90, CWN92, HJ91]. To illustrate this, Figure 1.1 plots trends in processor and DRAM performance from 1980 to 1992 [HP90]. The data (from 1990) indicate that high-performance processors have been improving by a factor of 100% per year since 1985 and extrapolated out to 1992. However, improvements in DRAM speeds have not kept pace; since 1980, row access times for DRAMS have improved at a rate of only about 7% per year.

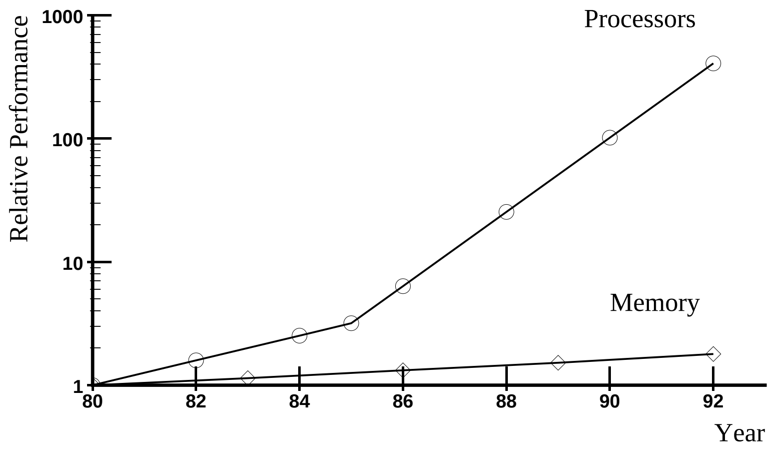

Figure 1.1: Processor and memory performance trends: 1980 to 1992.

The result is that main memory latencies in systems built with current generation high-performance microprocessors are typically quite large. In today's high-performance workstations, cache miss latencies of 20 to 50 cycles are not uncommon. By comparison, on the VAX 11/780 computer (shipped in 1978), the additional time required to service a cache miss was less than a single instruction execution time [HP90].

In current shared-memory multiprocessor machines, latencies can be even larger. As an example, the Stanford DASH multiprocessor [LLJ$^+$93] arranges its processors in clusters interconnected by a mesh. For references in which the data must be fetched from a remote memory in another cluster, DASH has memory latencies of roughly 130 processor cycles. These higher multiprocessor memory latencies result both from the physical distribution of the memory and from the potential contention between processors on the shared interconnection network.

1.1.1 Using Memory Hierarchies to Mask Latencies

The most common technique for mitigating the effects of the processor-memory performance gap is to introduce one or more levels of caches, as shown in Figure 1.2. A small, fast, cache memory placed directly on the processor chip can feed the processor with data, and can in turn be fed either by a larger (but slower) cache, or by main memory. Cache hierarchies become even more important in multiprocessors, because of the typically larger memory latencies. That is, since a cache miss serviced by a remote processor can incur over 100 cycles of delay, it becomes especially important that references to commonly used data can mainly be serviced by the cache.

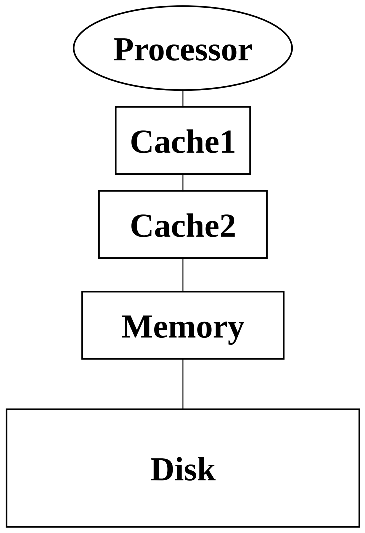

Figure 1.2: Uniprocessor memory hierarchy.

Caches improve the memory performance encountered by the application by taking advantage of several types of program locality. In programs with good *temporal locality*, once a data item has been referenced, it is very likely to be referenced again soon. Caches take advantage of this by fetching data items when they are first referenced and storing them in fast memory until they are replaced. As long as data are in the cache they can be accessed with low latency.

In programs with good *spatial locality*, once a data item has been referenced, other nearby items are likely to be referenced soon. Caches take advantage of this by loading not just the referenced word, but a group of multiple words (also called a *line* or *block*) when a cache miss occurs. Other items in the same line can be subsequently referenced with low latency. (We refer to this effect as *cache line prefetching*.)

In multiprocessors, programmers also try to exploit an additional dimension of locality. Since operations involving remote processors or remote memories can involve time-consuming transactions across a network, programmers often schedule tasks to localize memory requests to a particular processor. In this way, cache misses can be reduced. Furthermore, by appropriately placing data in local memories, misses are more likely to be serviced by local memory, rather than remote memory, when they do occur. When programs exploit locality in all of these ways, caches can be extremely effective at masking the high main memory latencies.

1.1.2 Improving Cache Performance

Unfortunately although locality is essential to good application performance, many programs fail to fully exploit it. In some cases, intrinsic characteristics of the algorithm preclude localized memory references. In other cases, poor application memory performance is simply a result of the specific coding implementation. In either case, when a tool gives profile information showing that memory stalls limit program performance, it can induce the compiler or programmer to restructure the algorithm or implementation so that it takes better advantage of the memory hierarchy.

There are several primary categories of performance “bugs” that memory profilers can point out to programmers. For example, *cache interference* arises from interleaved references to different data structures which map into the same region of the cache. Another potential problem, *poor spatial locality*, refers to cases where a program’s sparse or irregular access patterns do not take advantage of cache line prefetching. Finally, parallel code is subject to the same sorts of problems as sequential code, but in addition can also have memory bottlenecks due to *inter-processor communication* and *excessive sharing*.

Since a large program can have many potential causes of poor memory performance, detailed program information is needed to discern *where* and *why* memory bottlenecks are actually occurring. Some of this program analysis can be performed statically. For example, simple static compiler analysis can detect some instances of poor spatial and temporal locality, and use techniques like *loop reordering* and *data blocking* to transform access patterns to improve the memory referencing patterns. However, the scope of such analysis is limited mainly to scientific, loop-oriented code. When looking at a broader class of applications, more general analysis, information, and transformations are needed.

Previous Performance Monitoring Approaches

One can look to previous performance monitoring tools to see some examples of more general performance analysis, but very few performance monitoring tools have been introduced specifically to identify where *memory* bottlenecks are occurring. One example, Mtool [Gol92], provides information on *where* memory bottlenecks are occurring by presenting per-basic-block statistics on the amount of time spent on memory stalls. However, tools providing information at this level offer no insights as to *why* bottlenecks are occurring. Without more detailed information on why bottlenecks occur, it is often difficult to discern the problem and develop a strategy for remedying it.

Moving in this direction, SHMAP [DBKF90] is an example of a performance monitoring tool which does provide some detailed information on application memory behavior. SHMAP presents a reference-by-reference animation of the actual program memory behavior. This animation of cache activity can allow programmers to discern memory performance pitfalls. However the tool relies on programmers observing long animations; it offers little summary information or support for automatically analyzing the memory behavior. Especially in irregular, non-scientific code, reference patterns and performance bugs may be difficult to understand through this almost purely visual (non-numeric) approach. In any case, the volume of animation data required to analyze a real benchmark is overwhelming.

An additional limitation of approaches like SHMAP is that for collecting application reference traces, they rely either on expensive specialized monitoring hardware, or on simulation-based approaches. Monitoring hardware limits the generality of the tool, since most production machines do not provide trace capturing hardware. On the other hand, simulation-based approaches are more general, but have previously been too slow to encourage the programmer to iterate through several program tuning steps. In this work we address these limitations by developing MemSpy, an efficient simulation-based implementation of a detailed memory performance profiler.

1.2 Thesis Statement

This thesis argues that *detailed* information on a program's dynamic referencing behavior is necessary to tune many memory performance bottlenecks. Furthermore, it is natural to understand the interactions of different data structures and different code segments by viewing statistics in terms of *both* data and code structures in the program, rather than solely in terms of code structures. Performance monitoring tools can be crucial in automating the collection and presentation of such program performance data. We implement a tool, MemSpy, based on this premise.

Since efficiency is a key concern in such simulation-based performance monitoring, we introduce and evaluate a set of optimizations which reduce the execution time overhead required to gather such information. Thus, we are able to give detailed memory statistics about a program at speeds that are competitive with other, less detailed approaches.

1.2.1 Contributions

This dissertation makes the following contributions:

- **Data Oriented Statistics**

  This dissertation introduces the notion of *data oriented statistics*. Here, information is collected and presented in terms of source level data objects with which the programmer is familiar. Such information is orthogonal to traditional code oriented techniques, and combining the two offers powerful new views of program behavior.

We have found that data oriented statistics can be a natural way of viewing memory bottlenecks, since the way a program accesses memory is so intertwined with the data structures it uses.

- **Detailed Memory Performance Statistics**

  This work also shows the importance of detailed statistics, such as information on the *causes of cache misses* and *causes of cache replacements*, when tuning memory system behavior. Such information allows programmers to understand *why* bottlenecks are occurring, in addition to *where* they are occurring. Typical sources of poor memory performance, such as poor spatial locality, cache interference, and inter-processor sharing, are most easily distinguished by noting the exact causes of the resulting cache misses. When programmers must select from many possibly useful ways of tuning code, information like this guides them towards the transformations that will be most fruitful.

- **Simulation Performance**

  Statistics at this level of detail are difficult to collect except by software simulation; however tools based on memory system simulation have previously been dismissed as being impractically slow. This work refutes that belief by implementing a tool, MemSpy, which uses an efficient memory simulator to gather the data described above. We present an optimization, called *hit bypassing*, which specializes the simulation of cache hits, leading to significant performance improvements. With it, MemSpy's simulation based approach can be implemented with roughly a factor of 8 to 17 times overhead for sequential programs, and roughly 30 to 50 times overhead for parallel applications.

- **Reference Trace Sampling**

  Finally, to further optimize simulation performance, this thesis examines the use of reference trace sampling. Unlike hit bypassing, performance optimizations from sampling explicitly trade off small decreases in simulator accuracy for significant performance improvements. Using sampling, cache miss rates can be estimated to within 10% of their true value while simulating only about one tenth of the total references of the program. Sampling further improves MemSpy's performance, bringing the final execution overhead down to factors of 3 to 10 for sequential code, and roughly 8 to 25 for parallel code. That is, sequential programs that normally run in 1 minute will take only 3 to 10 minutes to run when generating MemSpy statistics. Since MemSpy's detailed statistics can actually accelerate tuning by giving programmers important insights on program bottlenecks, this overhead is generally quite acceptable.

1.3 Organization

In Chapter 2, we begin with a discussion of common program characteristics that result in poor memory system behavior. We then discuss the kinds of information that would be useful in isolating and understanding such problems. Following this, in Chapter 3, we present two case studies that demonstrate the use of such statistics to tune both sequential and parallel applications. These case studies illustrate MemSpy's discovery of memory performance bugs involving cache interference and poor spatial locality.

The case studies of Chapter 3 raise interesting questions of how one determines appropriate divisions for data oriented statistics, and how one efficiently implements such detailed statistics gathering. Thus Chapter 4 presents the design details related to collecting this information. We also present heuristics for aggregating information on the individual data structures together, and the naming issues in applying labels for these statistics aggregations.

Having demonstrated the utility of MemSpy's memory performance profiles and discussed MemSpy's implementation, in Chapter 5 we go on to discuss the tool's execution time overheads in gathering this data. This chapter introduces performance optimizations which further reduce the execution time overhead for simulation based tools such as MemSpy.

While the tool performance overhead after the initial optimizations in Chapter 5 is already quite good, Chapter 6 describes an additional performance optimization: reference trace sampling. We present results on accuracy issues in implementing sampling for both sequential and parallel applications. We then discuss the significant performance benefits available through sampling.

Chapter 7 discusses related work in the areas covered by this dissertation. Finally in Chapter 8, we offer conclusions and suggest avenues for future work.

To illustrate MemSpy's use on a broader set of performance tuning examples, Appendix A presents additional case studies using MemSpy. Finally, Appendix B presents a full description of the statistics provided by MemSpy.

Chapter 2

Tuning Program Memory Behavior

As the relative speeds of processors and memory have diverged, memory system performance has become an increasingly important factor in achieving good overall program performance. The goal of this chapter is to give an understanding of ways in which programs fail to use caches effectively, and the information required to isolate these problems and reason about them.

We first describe the primary types of memory performance "bugs" in sequential and parallel programs. This description in Section 2.1 also discusses the information and statistics that programmers and compilers need to identify such performance bugs. This drives the discussion in Section 2.2 of specific performance monitoring features we propose for identifying where and why such bugs are occurring. Building on this, Chapter 3 gives case studies that demonstrate these features as implemented in MemSpy.

2.1 Common Memory Performance Bugs

On modern computers it is essential that programs exhibit good memory reference locality, in order to make good use of the caches and achieve high performance on these machines. In reality, however, programs often access memory in ways that that *do not* make effective use of the memory hierarchy. The goal of this section is to outline the main ways in which applications can fail to obtain good performance from cache systems. These performance "bugs" stem from combinations of both (i) unfavorable referencing characteristics in the programs (such as a lack of spatial or temporal locality) and (ii) implementation constraints on the cache designs (such as limited cache associativity).

For each performance bug, we discuss the characteristics that can lead programmers to discover the problem, and the methods for tuning it once it has been diagnosed. We use this section's characterizations of memory performance bugs and their tuning techniques to drive the next session's discussion of what information is called for in performance debugging tools.

2.1.1 Cache Interference

The first performance bug we discuss is cache interference, which occurs when multiple memory lines map to the same line in the cache. Thus interleaved references to these lines may lead to conflicts in the cache, causing premature replacements of useful data.

As an extreme example of cache interference, consider a program that is performing a vector addition on two vectors positioned such that they map into exactly the same lines of the cache. As the program references corresponding vector elements, the elements will interfere in a direct-mapped cache, since memory lines for both vectors will map to the same cache line. This interference can lead to much larger amounts of memory stall time than if the vector elements did not interfere in the cache. For example, if four vector elements fit on a line, then without interference, cache misses need occur only once every four references to each vector, because the lines could remain in the cache between references. With interference, the lines will be replaced out of the cache prematurely, and cache misses may occur on each reference to a vector element.

We find that cache interference is most prevalent when caches have low associativity, especially in direct-mapped caches that are commonly used for primary data caches in current processors. It is also more significant in smaller caches, where the program data space "wraps around" the cache more often. For a data space of a given size, mapping it into a smaller cache increases the number of memory lines that map to each cache line. This increases the probability of a bad alignment between two data structures.

Cache interference is strongly tied to the way the data structures map into the cache and is clearly a function of the positioning of data structures in memory. Because of this, it can be hard for programmers to predict or reason about, since high level languages often shelter programmers from details like the precise positioning of the program data.

Recognizing the Problem

Tools can help programmers recognize memory performance bugs by providing particular performance information. First, leaving cache interference aside, tools can assist in diagnosing and tuning any memory performance bug by pointing out data structures or sections of code where the memory stall time attributed to it is both “large” in an absolute sense, and “larger than expected” in a relative sense. That is, if a data structure has a “large” memory stall time, but this stall time is expected and unavoidable, then this is not a performance bug. Conversely, if the data structure has a “larger than expected” share of the stall time, but its share is not significant to program performance in an absolute sense, then the performance bug is of little consequence. While the tool often cannot directly point out “larger than expected” stall times, programmers can use the size of the data structure and the access pattern to reason about the expected memory behavior.

Beyond this, the specific bug of cache interference is recognized by seeing that the cache misses are primarily caused by replacements. That is, data items were previously in the cache, but were prematurely replaced out of the cache before the re-reference that caused the miss.

To fully understand and eliminate a problem of cache interference, one also needs to ascertain whether the problem is one of *cross-interference* or *self-interference*. In cross-interference, the memory corresponding to *multiple* data structures is conflicting in the cache, leading to interference. In self-interference, different portions of the *same* data structure interfere in the cache. Tools can help programmers distinguish these two cases by providing information on the data structure causing the cache replacements. Information on the causes of replacements is important because it guides the programmer towards an appropriate tuning fix.

Tuning the Memory Behavior

Once cache interference has been diagnosed, program transformations can be used to reduce or eliminate the effect. In some cases the interference occurs because of a specific alignment between two data structures. In such cases, the interference can be reduced by adding additional code that checks the alignment of the two data structures when they are allocated, and offsets one data structure slightly if needed to avoid cache conflicts.

In other cases, interference sometimes occurs because the referenced memory regions are not stored contiguously, and thus are more likely to wrap-around in the cache and possibly interfere. For example, in the blocked matrix multiply case study in Chapter 3, non-contiguously stored rows in the sub-block of the matrix interfere with each other in the cache. By copying the current sub-block so it is contiguously stored in a separate buffer, the probability of wrap-around is reduced, and the incidence of cache interference is diminished as well.

2.1.2 Poor Spatial Locality

A second class of memory performance bugs stems from a lack of spatial locality in the application reference behavior. In programs displaying good spatial locality, once an item is referenced, items nearby in memory will also tend to be referenced soon. Caches take advantage of this by fetching data in multi-word lines, rather than fetching single words as they are accessed.

For example, unit stride accesses through a vector show very good spatial locality. They take advantage of *cache line prefetching* because a miss to the first element in a particular line brings an entire line of elements into the cache, and subsequent references to elements on the same line are more likely to be hits.

Examples of poor spatial locality fall into several categories. An extreme example of poor spatial locality is the case of accessing a two dimensional array in row-major order when its elements are stored in column-major order, or vice versa. More generally, non-unit strides through data structures reduce spatial locality compared to unit stride accesses. This is because a series of non-unit stride accesses will not touch as many of the data items on a single cache line before moving on to the next cache line.

Spatial locality can also be degraded when data items referenced close to each other in time are stored as non-contiguous data structures. For example, in the Vrender application in Appendix A, three arrays are used to hold three different attributes about each point in a two dimensional space. An element of each of these arrays is accessed on each iteration of a loop, but between iterations the values are likely to be replaced from the cache. By merging the three arrays into a single data structure with three attributes per array element, all positioned on the same cache line, the programmer was able to significantly improve the spatial locality of the application. As these examples show, spatial locality is of primary importance when a data structure does not fit in the cache or is unlikely to remain in the cache between usages. When a data structure remains in the cache and is referenced several times, poor spatial locality on the references that bring it into the cache can be offset by good hit rates on subsequent references.

The problem of poor spatial locality often becomes even more pronounced in parallel code. When parallelizing code, data structures that were stored and accessed contiguously in the sequential case can become distributed over several processors in the parallel case. Thus, even though the data structure as a whole may be stored in a contiguous region in shared memory, each individual processor may use non-contiguous regions. The Tri example in Chapter 3 gives an example of such a performance bug, and a reordering heuristic that was used to fix the problem.

Recognizing the Problem

As with cache interference, the first step towards recognizing poor spatial locality is to notice a data structure making a large contribution to the total program stall time. Although this does not always indicate a performance bug (it could simply indicate unavoidable memory stalls for this data structure), it does indicate places where program tuning, if needed, will be most fruitful. Beyond this, poor spatial locality can sometimes be diagnosed due to a large number of first reference misses for a particular data structure. However, if the poor spatial locality occurs in a phase after the data has been initialized, then the causes of the misses may not be indicative of the problem.

One piece of information that may be useful in identifying poor spatial locality is information indicating what fraction of data words on a particular line were touched before the line left the cache. This would give an indication of the effectiveness of cache line prefetching in pulling in data items that are actually used later. Unfortunately, this is not a foolproof approach for identifying poor spatial locality, since instances of cache interference or processor sharing (defined in the next subsection) can also cause a line to be removed from the cache before multiple words have been referenced from it.

Finally, information on reference stride can also be helpful in isolating some cases of poor spatial locality. An example of this might be statistics on the stride distance and data structures involved in adjacent accesses, accesses that are two references apart, three references apart, and so on. By providing summarized averages of stride information, users may be able to glean information on (i) strided accesses within a data structure, as well as (ii) typical interleavings of references to different data structures. The former information could be useful in determining when loop transformations or data reorderings within a data structure are warranted. The latter could be useful in determining when disjoint data structures could be merged for better performance.

Tuning the Memory Behavior

In general, one can say that poor spatial locality occurs when a program's data access order is not highly correlated with the data storage order. Thus, tuning fixes for spatial locality generally involve either reordering data accesses to match storage order, or reordering data storage to match access order. Techniques which reorder data accesses include compiler transformations like loop reordering and data blocking. Techniques which reorder data storage include the array merging technique used in Vrender (Appendix A) and the matrix row reordering heuristic used in Tri (Chapter 3). When the reorderings are feasible, they can provide for a better match of access and storage order leading to good memory performance.

2.1.3 Interprocessor Sharing

Finally, we discuss cache misses due to interprocessor sharing, one of the leading causes for poor memory system performance in shared memory parallel programs. In shared memory multiprocessors, multiple processors may each simultaneously hold the same memory item in their local caches. Cache coherence protocols are used to maintain consistency among the data cached by each processor, by requiring update or invalidate operations to be performed after one processor modifies a particular cache line. In these operations, other caches in the system are notified of the write operation, and if they are caching the same memory line, they update its value or invalidate it from the cache. In this dissertation, we primarily treat the invalidation based coherence model since it is more commonly implemented in production multiprocessors. However, excessive interprocessor sharing exacts a performance cost under both update and invalidate protocols.

Interprocessor sharing can be categorized as either *true sharing* or *false sharing*. The first category, true sharing, occurs when multiple processors are actively reading and writing the same data item. Since shared data is the basic vehicle for interprocessor communication in shared memory parallel programs, memory stalls due to sharing are often unavoidable. Programs can often be restructured, however, to minimize the communication required. For example, communication in parallel programs can often be reduced by assigning parallel tasks to processors in ways that localize data usage to particular processors. True sharing becomes especially significant in large-scale multiprocessors, because in these machines, cache misses following an invalidation can incur memory latencies of one hundred cycles or more.

The second category, false sharing, refers to the situation where multiple processors are actively reading and writing *different* variables on the same cache line. For example, this can happen when distinct elements of an array are used by different processors, but stored on a single cache line. Even though the processors may not be writing to the same array elements, coherence protocols will be executing transactions to keep the line's value consistent in all caches. False sharing becomes increasingly important with the general trend towards longer cache lines in multiprocessors, because of its dependence on the number of data items in a cache line.

Recognizing the Problem

Excessive sharing is recognized by noticing data structures with a large number of misses caused by invalidations. By separating information in a data structure oriented way, and providing information on the causes of cache misses, tools can greatly help in recognizing performance bugs like this.

To distinguish true sharing from false sharing, one needs further information that identifies the cause of the invalidation. Recall that false sharing occurs when a processor's copy of a cache line is invalidated even though the processor is not using the specific address from the cache line that caused the invalidation. Thus, false sharing can be distinguished from true sharing by tracking the bytes in the cache line that have actually been referenced by a particular processor, and comparing these to the address causing the invalidation. If the processor has not referenced the "invalidating" address since the line has been in the cache, then the invalidation is due to false, rather than true, sharing.

Tuning the Memory Behavior

Performance tuning for true and false sharing requires different approaches. Reducing true sharing is typically accomplished by restructuring the algorithm or implementation to localize the usage of particular memory regions to particular processors. For example, in the parallel version of Vrender discussed in Appendix A, the programmers assign particular portions of the image data to the same processor for computation in each of several iterations.

A key tradeoff in reducing true sharing is the balance between improving data locality and improving task load balancing. For example, when the amount of processing time required by particular parallel tasks is not known in advance, parallel programs typically distribute tasks dynamically to processors to balance the work done. Constraining task distribution in order to enforce data locality may degrade the performance through load imbalances more than it improves the performance through locality. One of the contributions of dynamic program monitoring tools is that they allow the user to actively test and compare different approaches, to achieve a balance between load balancing and data locality.

In tuning false sharing, one generally attempts to reorder either accesses or storage such that a particular cache line is mainly accessed by a single processor. In some cases this may involve high level algorithm or implementation changes to localize the task assignments. In other cases this may involve merging data structures (such as described in the LocusRoute example in Appendix A), to group together variables used by particular processors. Finally, when neither of these approaches is feasible, programmers or compilers can choose to pad variable definitions (for example, array elements) with extra dummy variables such that all the data on a particular cache line is likely to be accessed by only one processor, and no data used by other processors fits on that line. Of course the benefits of this latter approach have to be balanced against the drawback that it leads to overall increases in the program data space.

2.1.4 Summary

The goal of this discussion was to outline the primary categories of poor application memory performance, and discuss how to reorganize applications to fix them. While particular application characteristics may sometimes prevent tuning, some tuning fixes for these problems are quite trivial. Techniques like reordering loops to improve spatial locality or realigning data structures to reduce cache interference require little effort from the programmer, and can sometimes even be automated in the compiler.

In working with a collection of sequential and parallel applications we have found that even in cases when tuning a performance bug is not trivial per se, *diagnosing* the problem without proper tools is often more difficult than actually tuning it. For example, solutions like data structure merging in Vrender in Appendix A are relatively easy once the particular problem (in this case poor spatial locality) is identified by MemSpy. We see this as a strong argument for the development of detailed, data oriented tools capable of isolating and analyzing memory performance bugs.

2.2 Information for Tuning Memory Bottlenecks

Overall our experiences tuning sequential and parallel applications have shown that the key issue in developing performance tuning tools is to be able to indicate to programmers *where* the performance bottlenecks are and *why* they are occurring. From this information, programmers can often restructure or transform the code to improve the memory performance.

The following three subsections discuss the features we propose for locating and understanding memory performance bugs in sequential and parallel programs. These are (i) data and code oriented statistics, (ii) statistics on the causes of cache misses, and (iii) hierarchical data presentation.

2.2.1 Data and Code Oriented Statistics

To provide information on *where* performance bottlenecks are occurring, MemSpy provides high-level performance statistics broken down into both data oriented and code oriented subsets. *Data oriented statistics* are statistics presented in terms of source-level application data structures. Traditionally, performance tools have provided statistics only in terms of code structures such as procedures or basic blocks.

Figure 2.1 gives an abstract illustration of possible code oriented and data oriented subdivisions. While code oriented statistics only divide program statistics along one dimension, the key contribution of data oriented statistics is that they allow for statistics to be presented along a second dimension of this space, by subdividing the program into different data divisions. Data oriented statistics can be crucial to reasoning about memory behavior, and combinations of data and code oriented statistics are often instrumental in isolating particular memory bottlenecks.

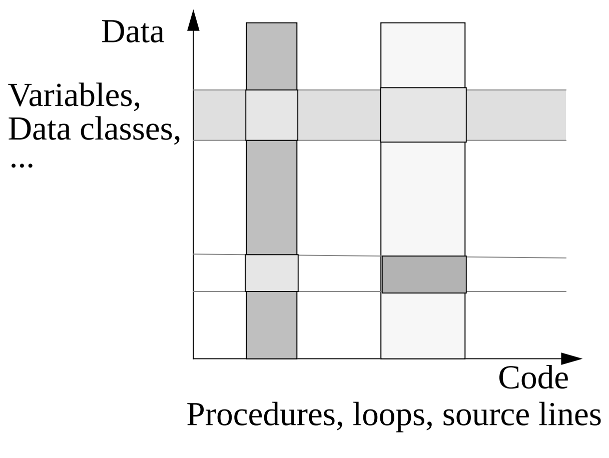

Figure 2.1: Decomposing programs into data and code statistical bins.

Fundamental design issues for data oriented (as well as code oriented) statistics lie in determining a natural granularity at which to present application statistics. Chapter 4 will discuss these implementation issues, the mapping techniques we propose for organizing these statistics, and the naming issues that arise in our approach.

2.2.2 Memory Statistics on Causes of Cache Misses

After determining where program bottlenecks lie, users need progressively more detail to understand *why* bottlenecks are occurring. Detailed statistics summarizing the frequency and causes of cache misses are quite useful for understanding and fixing memory bottlenecks.

As discussed in Section 2.1, cache misses occur for one of three reasons. First reference misses occur when a particular memory line has not been referenced before, and thus is not in the cache. Replacement misses occur when a particular memory line has been referenced before, but has been replaced out of the cache by another line. Invalidation misses occur in parallel programs when a particular memory line has been referenced before, but was invalidated out of the cache by another processor. MemSpy's statistics on the frequency and primary causes of cache misses, the programmer can often gain insight about the type of memory performance bottleneck present, and how to tune it.

In spite of the importance of detailed information in understanding such effects, most existing tools have provided no statistics on memory system behavior at all. Other tools, such as Mtool [GH91b], give only high-level information about which parts of the code cause memory bottlenecks. The lack of detailed support for memory bottleneck identification is partly due to the difficulty in efficiently gathering memory system statistics. Gathering detailed memory statistics requires fine-grained monitoring using either specialized hardware or software simulation. The drawback of specialized hardware is that it can limit the generality of the tool, but on the other hand software simulation can often be too slow. A major contribution of this dissertation is its proposal of optimizations that greatly improve the efficiency of simulation-based monitoring.

2.2.3 Hierarchies of Presentation

Because tuning memory behavior requires a large volume of detailed statistics, it is also important that the tool control the amount of information presented to the user at each step of the tuning process. Hierarchical statistics presentation is a common method for accomplishing this, and this section discusses two methods for making use of hierarchies in performance profiles.

Hierarchy of Detail

One hierarchy commonly used by tools is a *hierarchy of detail*. That is, as the user steps through the user interface, the statistics presented at each step move from high level to more detailed. Because MemSpy offers more detailed statistics than previous tools, it is able to offer a deeper hierarchy of detail than previous tools.

For example, MemSpy initially provides high-level information indicating which procedures in the program are responsible for most of the execution time. It also gives a breakdown of how much of this time is spent in computation, memory stalls, or (in parallel programs) synchronization. From this high-level overview, programmers can choose to bring up more detailed statistics. These detailed statistics include information on memory stall time, more detailed information on the causes of cache misses, and even more detailed information on the causes of replacements.

Hierarchies of Focus

Another common hierarchy is a *hierarchy of focus*. That is, at each step the user's attention is focused in on increasingly smaller portions of code or data. A significant aspect of MemSpy's design is how its data oriented statistics make available *several* new hierarchies of focus, in addition to the one available in code oriented tools.

In code oriented tools, it is common to provide statistics on the whole program's behavior, and then to allow the user to focus in on particular procedures, and perhaps even particular source code lines or basic blocks. The leftmost hierarchy in Figure 2.2 shows this sort of approach. Within a code structure like a procedure, one can break down statistics by smaller code units, like basic blocks or source code lines.

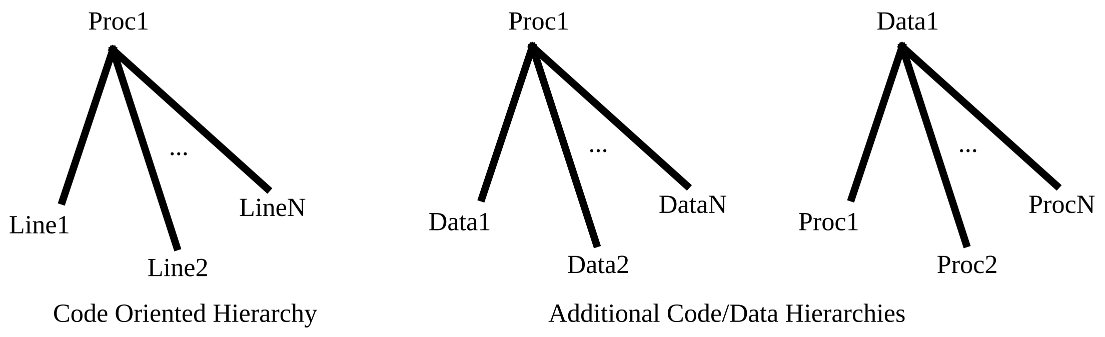

Figure 2.2: Hierarchies of focus in data oriented tools.

When one moves to an approach that is both data and code oriented, *additional* hierarchies of focus are available, such as the ones shown on the right in Figure 2.2. For example, within a particular *procedure*, one may want to separate the memory performance bottleneck according to which *data structures* incurred stall time. Moreover, data oriented statistics also allow the user to first view the effects of particular *data structures*, and then subdivide these effects by code structures such as procedures. This type of program view is especially useful in cases where a performance bottleneck is most naturally attributed to a particular variable, but has accesses spread over several procedures.

2.3 Chapter Summary

To summarize, this chapter has argued for the collection of detailed, data and code oriented memory performance statistics. It describes three major causes of poor application memory performance and how they can be recognized and tuned. We argue that performance monitoring tools employing (i) data oriented statistics, (ii) detailed statistics on the causes of cache misses, and (iii) hierarchical presentations, can be instrumental in identifying bugs and their causes. Performance information broken down by data structure is important in initially isolating memory performance bugs. In addition, information on the causes of cache misses and cache replacements can be invaluable in further understanding particular memory bottlenecks. Finally, a hierarchical presentation style can be extremely useful in organizing and managing the large volumes of information gathered. To support these claims, the case studies in Chapter 3 will concretely illustrate the information collected in our implementation of MemSpy, as well as MemSpy's use in tuning sequential and parallel applications.