Shared-Memory Performance Profiling

Zhichen Xu James R. Larus Barton P. Miller
{zhichen,larus,bart}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706-1685

Abstract

This paper describes a new approach to finding performance bottlenecks in shared-memory parallel programs and its embodiment in the Paradyn Parallel Performance Tools running with the Blizzard fine-grain distributed shared memory system. This approach exploits the underlying system's cache coherence protocol to detect data sharing patterns that indicate potential performance bottlenecks and presents performance measurements in a data-centric manner. As a demonstration, Paradyn helped us improve the performance of a new shared-memory application program by a factor of four.

1 Introduction

Distributed Shared Memory (DSM) alleviates some of the difficulty of programing a parallel computer by hiding the details of communication. The abstraction of a shared address space, while convenient, can also hide serious communication bottlenecks that cripple a program's performance. To effectively use shared memory, programmers need performance tools capable of piercing this abstraction and relating a program's behavior to the underlying hardware's actions.

This paper describes a new technique for collecting and displaying shared-memory performance information. The first key aspect of this approach is to detect cache block sharing patterns that indicate potential performance bottlenecks. Our current system detects these patterns at low cost using a modified cache coherence protocol for the Blizzard fine-grain distributed shared memory system [25] running on the Wisconsin Cluster Of Workstation (COW).[1] The other key aspect of our technique is to use the Paradyn Parallel Performance Tools [22] to present shared-memory performance data in a data-centric manner, which relates events on cache blocks to program data structures.

Shared memory programs communicate through references to shared data. The underlying system (hardware, software, or both) responds to a memory reference by communicating, to obtain a copy of data (which is typically cached for subsequent references) and to maintain the global state necessary to implement a coherent programming model. At a load or store, the actual communication depends on the memory system's protocols and the program's dynamic state in ways that can be difficult to understand or predict [9].

Nevertheless, detecting this hidden communication is essential to isolate and eliminate many shared-memory performance bottlenecks. For example, a cache block's migratory behavior may indicate false sharing, which a programmer can eliminate by aligning and padding data structures. Moreover, recent research has focused on new tools, such as custom protocols and extensible memory systems, that offer programmers far greater control over shared memory systems [3, 28]. For example, a programmer can greatly reduce the cost of producer-consumer sharing by increasing the cache block size or using an invalidate, rather than update, protocol [7]. Again, understanding a program's access pattern is the necessary first step to improving its shared-memory performance.

The first step in memory profiling is to detect access patterns that indicate performance problems. Our pattern detection mechanism is integrated in a shared memory cache-coherence protocol, so it incurs little overhead and requires no changes to programs. More importantly, this mechanism associates addresses with events, so a performance tool can present measurements at the program's level of abstraction. The performance tool can use these addresses to relate memory accesses to a program's data

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 (ARPA order no. B550), NSF Grant MIP-9625558, NSF NYI Award CCR-9357779, and Department of Energy Grant DE-FG02-93ER25176. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

To appear at the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Las Vegas, Nevada, 18-21, 1997.

1. The Wisconsin Cluster of Workstations (COW) is a collection of 40 Sun SPARCStation 20's (with dual 66Mhz Ross HyperSPARC processors) connected by a Myricom Myrinet and 100MB Ethernet.

structures and use conventional profiling techniques to connect accesses with the statements that execute them. We call this process *shared-memory performance profiling*. Memory profiling by itself cannot find all performance problems, so we built memory profiling into the more extensive facilities of the Paradyn performance tool.

Although this work exploits Blizzard's custom protocols, other systems can also provide mechanisms to associate shared memory communication with data. Any hardware shared-memory platform can support a fine-grain DSM system, like Blizzard [25] or Shasta [24], that exposes a coherence protocol for performance debugging. Since the underlying shared-memory hardware provides fast communication, a DSM system of this sort would incur only moderate overheads. Alternatively, a platform may provide hardware features, such as informing memory operations that trap on cache misses [12], which can be used to associate a program's memory references with a coherence protocol's actions.

This paper illustrates the use of memory profiling through an extended case study of tuning a new shared-memory protein folding code [13] from researchers in the University of Wisconsin Chemical Engineering Department. With the help of memory profiling and Paradyn, we improved this application's performance by more than a factor of 4 to an efficiency of 80% on 16 nodes.

The paper is organized as follows. Section 2 briefly reviews the features of Paradyn. Section 3 describes our approach to shared-memory profiling and our implementation techniques. Section 4 contains a case study that illustrates how Paradyn and shared memory profiling help identify and eliminate performance problems. Section 5 discusses automating the search for performance bottlenecks. Section 6 describes related work.

2 Paradyn Basics

This section briefly describe the basic characteristics of the Paradyn performance tools. Readers familiar with Paradyn can skip this section. Paradyn is a parallel performance measurement tool that currently runs on SPARC, Alpha, Power2, PA-RISC, and x86 platforms [22]. Paradyn is based on dynamic instrumentation, a technology that allows instrumentation code to be inserted, changed, and removed from a running application. Beside standard performance metrics such as CPU usage and blocking time, Paradyn can instrument other system, hardware, or network activity visible from an application program's address space.

Paradyn provides two basic abstractions: *metric* and *focus*. A metric is a time-varying function that measures some aspect of an application's performance, for example, CPU utilization or number of procedure calls. A focus is a component of a running application. Paradyn views a program as a collection of resource hierarchies that represent various elements of a program, such as code (modules, procedures), machines (hosts), or synchronization (messages, semaphores, locks). A collection of nodes in the resource hierarchy forms a focus. Tool users request Paradyn to collect metrics for the foci in which they are interested, and Paradyn dynamically instruments the program according to this criteria.

Figure 1 shows Paradyn's display of two resource hierarchies. The "Code" hierarchy appears on the left and contains modules in the program. Under each module is the procedures in the module. For memory profiling, we defined a new resource hierarchy ("Memory") that lists a program's major data structures. Under each data structure is a list of the cache blocks (identified by their memory addresses).

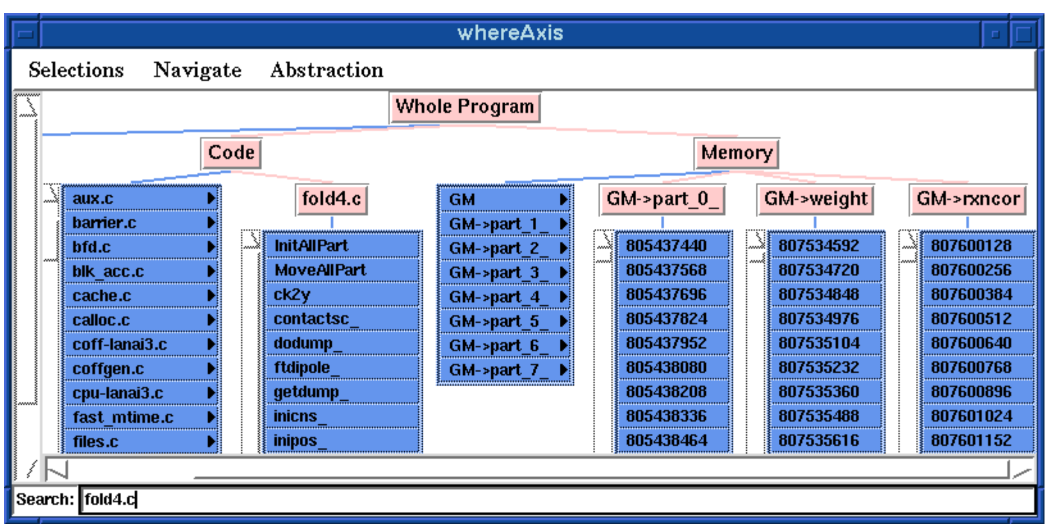

Figure 1: Resource Hierarchies in Paradyn.
"Code" and "Memory" hierarchies visible.

3 Shared-Memory Performance Profiling

The first step to alleviate a performance bottleneck is to detect it. In shared memory systems, many performance problems arise from excessive communication due to the interaction of a program's data accesses and the system's fixed cache coherence policy. For example, when a cache block contains values used separately by two processors, false sharing can cause a memory access to produce four or more messages. False sharing, like many shared-memory bottlenecks, is often difficult to detect statically, but is readily apparent at run time.

This section describes our technique for shared-memory performance profiling.

3.1 Pattern Detection

An obvious place to detect patterns in shared-memory communication is from within the coherence protocol that satisfies memory references. Most coherence protocols are expressed as a state machine driven by operations on a cache block. Some recent systems, including our Blizzard platform and commercial systems such as Sequent's NUMA-Q system [20], implement these state machines in software, which opens the possibility of introducing new states and actions to recognize access patterns indicative of performance problems. These states and actions should not change the behavior of the coherence protocol. Instead, they provide an efficient mechanism for recognizing patterns in transitions of a state machine.

Our pattern detection mechanism currently detects producer consumer, migratory, grouped, and read-only access patterns. Let **R** represent a memory read, **W** represent a write, subscripts distinguish cache blocks, and superscripts distinguish processors. A block's *home* node is the processor on which the block is originally allocated. This machine performs protocol actions for the block in many directory-based coherence protocols. The block's *owner* is the processor that currently has write access.

A migratory access pattern occurs when a block is accessed by different processors, in turn. For a given interval, only one processor accesses the block, so sharing is sequential, not concurrent. A coherence protocol sees migratory accesses as a sequence of events of the form: $(W^i)(R^i|W^i)^*(R^j)(W^j)(R^j|W^j)^*$ where $i \neq j$. To detect migratory sharing, our augmented protocol records a block's last owner at its home node. When a read or write request from another processor arrives, the augmented protocol updates the block's `lastOwner` field. When the home node receives a upgrade request (a write to a read-only block) and the number of sharers is one, the protocol compares the requestor against the `lastOwner`. If they differ, then the block appears to be migrating from `lastOwner` to the requestor, and the augmented protocol records that the block is migratory. The protocol clears this property if `lastOwner` is the same as the requestor or the number of sharers is not one.

A producer-consumer access occurs when a block is written by one processor and subsequently read by one or more other processors, and this process repeats. The home node sees the events: $(\mathbf{W}^i(\mathbf{R}^i|\mathbf{W}^i)^*(\mathbf{R}^j)^+)^*$ where $i \neq j$. As with migratory blocks, at a write miss to a block in the read-shared state, the augmented protocol compares the requestor against the block's `lastOwner`. If identical, the block is likely used in a producer-consumer relationship. More accurate information can be obtained by checking the block's sharer list against the requestor.

A group access occurs when a reference to a cache block is always followed by references to a collection of other blocks. Let **A** be an abbreviation for [**W**|**R**]. With this pattern, a home node repeatedly sees events of the form: $\mathbf{A}_i^x\mathbf{A}_j^x$.... To detect this pattern, the augmented protocol tracks, in a variable `lastAccessed`, the block that a processor last accessed. Each block records three extra fields: `groupId`, `guess`, and `accessInGroup`, which the algorithm in Figure 2 uses to detect group accesses.

```
If (currentBlock.guess == lastAccessed)
{
    currentBlock.groupId = lastAccessed.groupId;
    currentBlock.accessInGroup = true;
    lastAccessed.accessInGroup = true;
} else {
    currentBlock.guess = lastAccessed;
    currentBlock.groupId = address(currentBlock);
    curentBlock.accessInGroup = false;
}
lastAccessed = currentBlock;
```

Figure 2: Algorithm to Detect Group Sharing.

3.1.1 Implementation Alternatives

Augmenting a coherence protocol to collect additional information incurs a cost in both space and time (see Section 3.2.2). It is unrealistic to expect a parallel computer to use an augmented protocol as its default or for architects to build such a protocol in hardware. Fortunately, many recent distributed shared memory (DSM) systems implement cache coherence protocols in software [1,7,16,17,18,20,23,24], which offers advantages over hardware, not the least of which is the ability to support more than one coherence protocol. Software protocols also enable tools, such as Paradyn, to instrument and measure a protocol.

A less radical alternative is informing memory operations, which either set a condition code or trap on a cache miss [12]. These, or similar, operations can be used to detect protocol transitions in hardware-based shared memory systems. The performance instrumentation could maintain a shadow directory that tracks which processors have copies of each cache block. At a cache miss in the shared address space, the information in this directory suffices to determine which actions a coherence protocol would perform.

Another alternative, for systems with no hardware support, is to performance tune programs using a fine-grain distributed system, similar to Blizzard [23] or Shasta [24]. Although this software layer is redundant for hardware shared-memory systems, it exposes the coherence protocol to a performance tool. Moreover, hardware shared memory's low latency ensures that the software overheads of the DSM system will be moderate.

Performance counters in existing systems, for example Sun's UltraSPARC [27], record how many cache misses occur in an interval. However, these counters cannot fully support memory performance profiling, as they provide no information about which cache blocks incurred misses.

3.2 Memory Profiling

Memory profiling associates coherence activity with a program's data structures and code. It forms the necessary bridge between low-level coherence activities, which operate on cache blocks, and a programmer, who thinks at a higher level of abstraction. Memory profiling extended Paradyn in two ways. First, we added many new memory- and cache-specific performance metrics. Second, we added data-centric views for both existing and new performance metrics.

The new performance metrics include frequency and time for many coherence events, messages, and synchronization events. Figure 3 lists the new memory performance metrics. Since Blizzard's cache coherence protocols run in an application's address space, Paradyn collects these metrics with its usual instrumentation techniques.

Performance profiling typically associates a metric with a control structure in a program. For example, a tool reports CPU time or messages for a module, procedure, loop, or statement. The complementary *data-centric view* associates metrics with data structures and often provides new insights into a program's interaction with the memory system. Data-centric presentations have been used for distributed arrays in data-parallel languages [14,15] and in cache tools for sequential programs [8,19]. The data views that we added to Paradyn combine fine-grained profiling (down to individual cache blocks), scalability (large data structures and large programs), and low overhead. To present this data, we added a new Paradyn resource hierarchy for shared memory (see Figure 1). Shared-memory resources currently fit in a two-level hierarchy consisting of data structures and cache

Metric Name	Description	Metric Name	Description
targetBarrier	Count of application-level barrier ops	targetBarrierWall	Time at application-barriers
activeMessages	Count of active messages	activeMessageWall	Time for active messages
bulkDataTransfers	Count of bulk data transfers	bulkDataTransfer-Wall	Time for bulk data transfers
pageFaults	Count of page faults	pageFaultWall	Time for page fault handling
memoryBlockTime	Wall time spent on all coherence misses		
stacheReadMisses	Count of read misses on non-home nodes	stacheReadMissWall	Time for non-home read miss handling
homeReadMisses	Count of read misses on home node	homeReadMissWall	Time for home read miss handling
stacheWriteRO	Count of write misses on read-only blocks, non-home nodes	stacheWriteROWall	Time for write misses on read-only blocks, non-home nodes
homeWriteRO	Count of write misses on read-only block, home node	homeWriteROWall	Time for write misses on read-only blocks, home node
stacheWriteInv	Count of write misses on invalid blocks, non-home nodes	stacheWriteInvWall	Time for write misses on invalid blocks, non-home nodes
homeWriteInv	Count of write misses on invalid blocks, home node	homeWriteInvWall	Time for write misses on invalid blocks, home node
invRO	Count of invalidations on read-only blocks	invROWall	Time for invalidations on read-only blocks
invRW	Count of invalidations on writable blocks	invRWWall	Time for invalidations on writable blocks
Polls	Count of poll operations	SentMsgs	Count of messages sent
SelfMsgs	Count of messages sent to self	Acks	Count of acknowledges
MsgBytes	Count of message bytes	RecvMsgs	Count of messages received
BufMsgs	Count of messages buffered	PollMsgs	Count of poll messages

Figure 3: Blizzard/Paradyn Memory Performance Metrics.
New performance metrics for Blizzard cache coherence protocols.

blocks.

3.2.1 Implementation

Paradyn creates counters or timers to collect performance metrics for a cache block or data structure and uses dynamic instrumentation to insert code to update a counter or timer. Paradyn's data collection facility periodically reads values from these counters or timers and ships them to the Paradyn front end. For example, to measure the time to handle coherence misses, Paradyn inserts code that starts a timer when a fault occurs and stops the timer when the application resumes. To record performance statistics for a cache block, Paradyn allocates a counter or timer for the block, and inserts code into the coherence protocol to update the counter or timer at faults on the block. To monitor a contiguous data structure, Paradyn updates the structure's counter when a fault falls inside the structure's boundary.

A major challenge in memory profiling is handling the large number of cache blocks in shared-memory applications. These blocks form a much larger focus than the other ones that Paradyn typically handles. We reduced the overhead of shipping shared memory information to the Paradyn front end by compressing shared-memory information and batching several messages together.

3.2.2 Memory Profiling Overhead

A second challenge is to control the instrumentation overhead. Instrumentation in a protocol handler must discriminate between a monitored and unmonitored block, which becomes expensive as the number of monitored blocks increases. We optimized the instrumentation code, with a technique called "vectorization", in which Paradyn allocates a vector of counter/timers for all monitored cache blocks and indexes into the vector (instead of allocating single counters or timers and testing for individual cache blocks). Vectorization reduced the instrumentation overhead from a cost linearly proportional to the number of monitored cache blocks to a constant cost. The vectorization test has approximately the same cost as instrumenting three cache blocks with the simpler approach. Further details are reported elsewhere [11].

Figures 4 and 5 report the overhead of profiling cache blocks and data structures. The measurements are from version 3a of the protein folding application (see Section 4). Figures 4 and 5 show that, with vectorization, the instrumentation overhead of 2% is independent of the number of profiled cache blocks. Data collection overhead is linearly proportional to the number of counters and timers. The data collection overhead is about 9% for 240 counters/timers

Number of cache blocks	No instrumentation	Counter-based Metrics		Timer-based Metrics	
		Instrumentation only	With data collection	Instrumentation only	With data collection
10	129.3	131.5 (2%)	134.3 (4%, 2%)	132.4 (2%)	134.6 (4%, 2%)
20	129.3	132.1 (2%)	137.4 (6%, 4%)	132.2 (2%)	137.6 (6%, 4%)
40	129.3	132.2 (2%)	142.5 (10%, 8%)	132.1 (2%)	144.0 (11%, 9%)

Figure 4: Overhead of Profiling Cache Blocks with Vectorization.
6 counters or timers (i.e., 6 performance metrics) per cache block. The stache block size is 128 and the application ran on 8 processors. Times are in seconds. Numbers in parentheses are percent increase over "No Instrumentation" case.

Number of cache blocks	No Instrumentation	Counter-based Metrics		Timer-based Metrics	
		Instrumentation only	With data collection	Instrumentation only	With data collection
100 blocks	129.3	131.6 (2%)	132.6 (3%)	131.7 (2%)	132.6 (3%)
200 blocks	129.3	131.4 (2%)	132.6 (3%)	131.6 (2%)	133.0 (3%)
400 blocks	129.3	131.5 (2%)	132.8 (3%)	132.3 (2%)	133.0 (3%)

Figure 5: Overhead of Profiling cache blocks with Aggregation.
6 counters or timers (i.e., 6 performance metrics) per data structure. The stache block size is 128 and the application program ran on 8 processors. Times are in seconds. Numbers in parentheses are percent increase over "No Instrumentation."

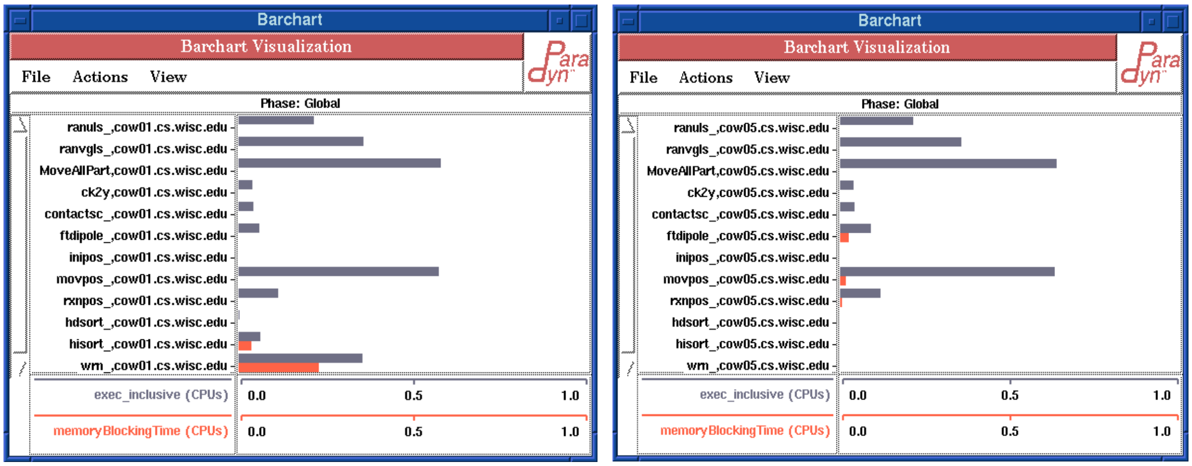

Figure 6: A Sequential Performance Bottleneck (version 1).

The bars label the resource being profiled. For example, "ranuls_,cow01.cs.wisc.edu" is the data for procedure "`ranuls_`*" running on host "*`cow01.cs.wisc.edu`*". The* `exec_inclusive`*" metric is the total elapsed time spent in a procedure and the procedures it calls. The "*`memoryBlockingTime`*" is the total time spent waiting for shared-memory delays.*

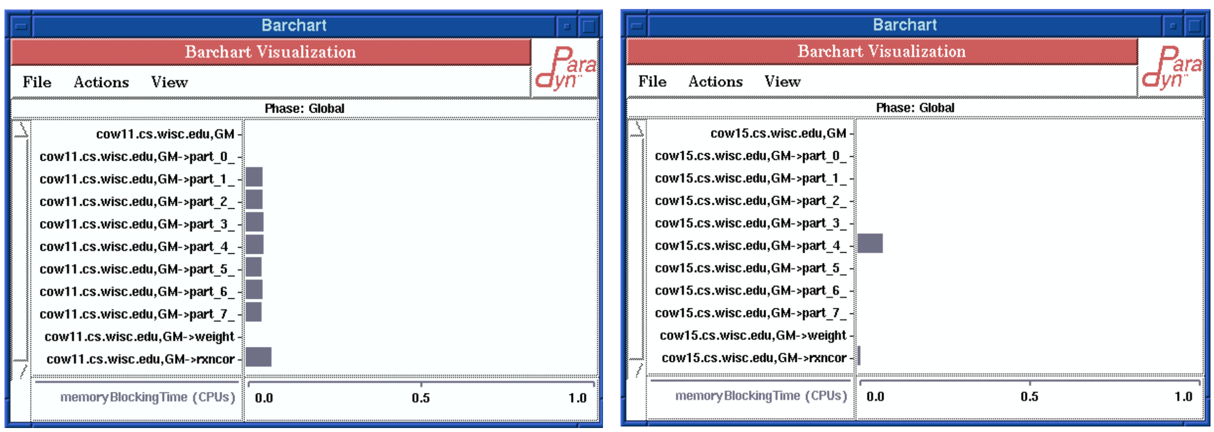

Figure 7: Blocking Time due to Coherence Misses (version 1).

(40×6).[2] Profiling an entire data structure is far less expensive than profiling its blocks separately, as instrumentation overhead is independent of the size of a data structure and data collection cost is proportional to number of counters or timers. Profiling a data structure with 6 metrics incurs an overhead of approximately 3%.

4 A Case Study

To illustrate how memory profiling can help identify and eliminate performance bottlenecks in a shared-memory application, we tuned a new parallel application with the aid of Paradyn. The application was written in the Chemical Engineering Department at the University of Wisconsin. It uses a weighted-ensemble Brownian (WEB) dynamics algorithm [13] to simulate protein association reaction (folding). The first version (version 0) of the application was written in Fortran 77 and ran on a SGI Power Challenge. Version 0 alternated sequential code with parallel loops (`doacross` with directives). We converted Fortran to C with `f2c`, modified it for the shared-memory programming model on the Wisconsin COW (Cluster Of Workstation), and started our performance tuning (version 1).

4.1 A Sequential Performance Bottleneck

The first performance problem that we encountered was that the serial portions of the application consumed a significant fraction of its total execution time. Much of this time was spent handling coherence misses. Figure 6 shows Paradyn's display of the execution times of a few important routines and the time spent handling coherence misses in each function. The left bar chart in Figure 6 shows node 0,which executes the sequential code. The right bar chart shows exe-

2. More recent versions of Paradyn use a new data collection technique based on shared memory that further reduces data collection costs.

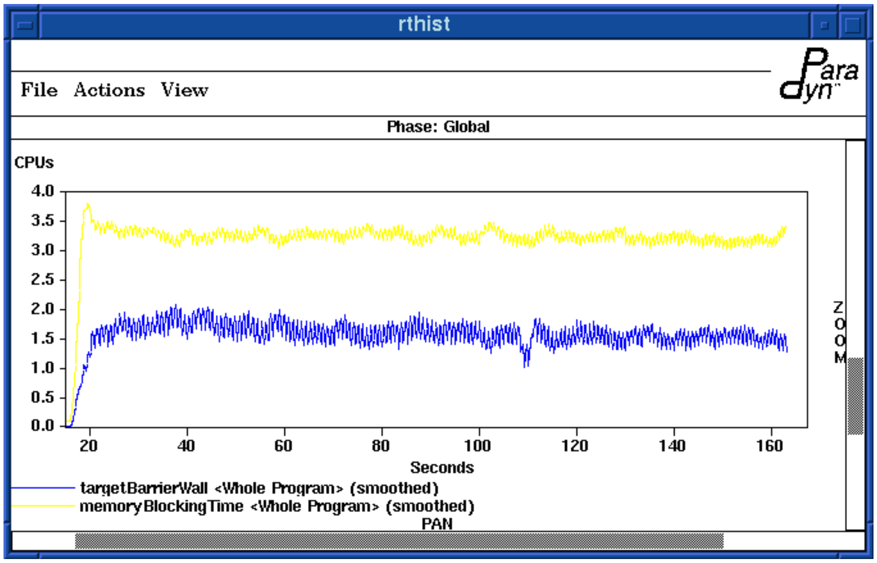

Figure 8: Memory Blocking and Barrier Wait Time (version 2).

cution and coherence miss times on another node. The behavior of nodes other than node 0 are similar to the right bar chart.

From Figure 6, it is clear that when the serial part runs on node 0 (bottom bars labeled `wrn_`), the other nodes sit idle. This serial code consumes about 40% of the total execution time on 8 nodes, and $^2/_3$ of this time is due to coherence misses (the lighter bar shows coherence miss time). Another observation from this figure is that coherence misses in the parallel phase (functions other than `wrn_`) are not as costly.

To find which data structures incurred these coherence misses, we associated coherence miss times with the major shared data structures in the application.The program uses three major shared data structures: `rxncor` tracks the reaction coordinates of all particles, `part` stores the orientation and position of the particles, and `weight` stores the particles' weights. The bar charts in Figure 7 show coherence miss times for the major data structures on node 0 and another node. The data structure labeled `GM->rxncor` is the data structure that stores the reaction coordinates. The data structure labeled `GM->weight` stores the weight of all particles, and the bars labeled with `GM->part_0_` to `GM->part_7_` are fragments of the particle data structure.

Figure 7 shows that references to the particle data structure causes the most coherence misses. In addition, node 0 misses on most parts except for the fragment labeled `GM->part_0_`, and node i $(1 \le i \le 7)$ misses on the fragment labeled `GM->part_i_`. Together, these facts suggest that the fragment `GM->part_i_` bounces back and forth between node 0 and node i. Profiling cache blocks belonging to `rxncor` revealed the same phenomena. Further evidence came from the pattern detection, which showed that cache blocks of `rxncor` have a migratory access pattern and cache blocks of `part` have both migratory and producer-consumer patterns.

4.2 Restructure the Folding Application

Examining the code showed that in the parallel phase of the application, each processor moved a portion of the particles. Then, the serial code sorted particles by their reaction coordinates and split or combine them to simulate a reaction. Although a custom protocol could transfer data more efficiently, we chose to restructure the application to eliminate this serial bottleneck. The performance data guided this design decision. After eliminating the coherence misses, the sorting, splitting and combining phase would still consume 15% of the total execution time (8 nodes), which would severely limit the program's speedup.

The modified code partitions particles according to their reaction coordinate and parallelize the serial phase, so each processor sorts, splits, and combines its particles. This change introduced an auxiliary data structure `tmpl` that tracks which processor owns a particle. The program after restructuring is called version 2.

Parallelizing the serial phase eliminated the sequential bottleneck and made computation and communication on different nodes symmetric. Although the change improved performance by a factor of 2, the program's speedup was still unsatisfactory (5.9 on 16 nodes). Memory profiling showed that considerable time was still spent handling coherence misses and waiting at the barriers introduced for the new synchronization. Figure 8 shows that for 8 nodes, 40% of the total time was spent handling coherence misses (memoryBlockingTime) and about 20% of the time at barriers (targetBarrierWall). To find the cause of the high memory blocking time, we examined the program's access

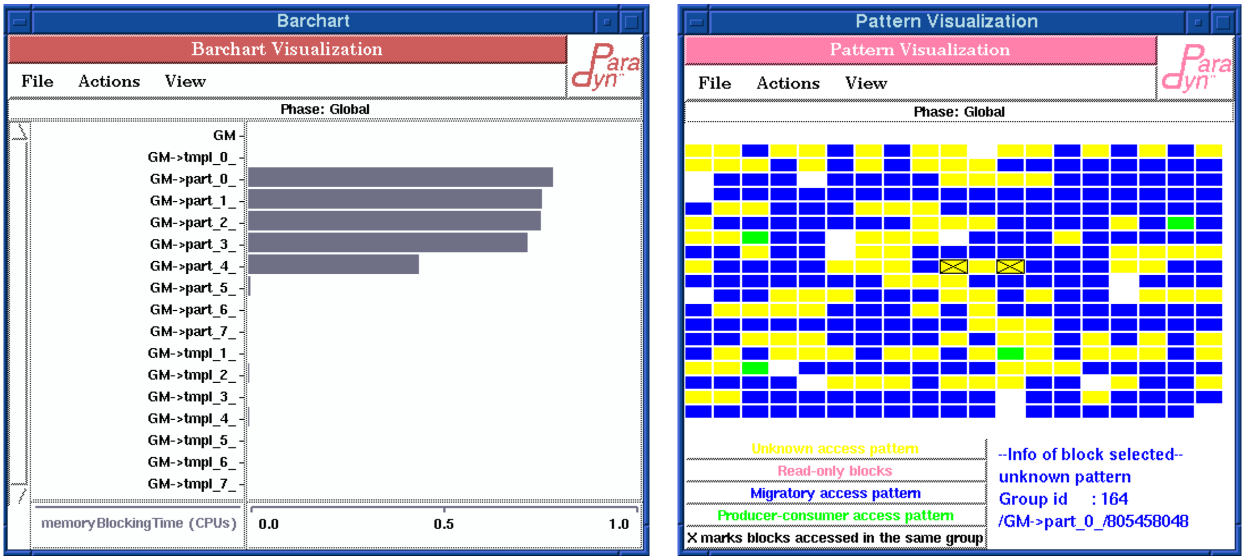

Figure 9: False Sharing in The Particle Data Structure (version 2).

In pattern visualization, the blue (darkest boxes in black&white) shows the migratory access pattern, green (medium gray in b&w) shows producer/consumer access, and yellow (light grey in b&w) shows an unknown pattern. The blocks marked with an “X” are a block selected by a user and the blocks accessed in the same group as the selected block. The text in the right corner shows information about the selected block.

patterns to see how the data structures were referenced. Figure 9 contains a visualization of the program's access pattern, which shows that the cache blocks in `part_0_` have a mostly migratory access pattern.

As mentioned earlier, a migratory access pattern can indicate false sharing. Examination of the code confirmed that this data structure was not cache-block aligned. To eliminate false sharing, we aligned and padded particles to ensure that each particle fell inside a single cache block. We call this version 3a. This alignment and padding drastically reduced memory blocking time and improved performance by a factor of 2. Figure 10 shows the time spent at barriers and coherence misses after alignment and padding.

4.3 Load Imbalance

Figure 10 shows that coherence misses are no longer a performance bottleneck. However, 20% of the program's time is still spent at barriers. The figure also shows that barrier time is not evenly divided among nodes, which suggests a load imbalance. As an experiment, we statically partitioned the particles, according to their processor's barrier waiting times (version 3b). Figure 11 shows that static load balancing reduced barrier and coherence miss time. Since particles move randomly, a dynamical load balancing scheme is necessary in general.

4.4 Summary

Figure 12 summarizes our performance tuning. It shows speedups of the different versions, including the original Fortran version on SGI Power Challenge. The initial version scales moderately up to 4 COW nodes, but its efficiency dropped drastically after that. The restructured program initially performed worse on a small number of COW nodes, because of the extra work to track which particle belong to which node. The program scaled better without the sequential bottleneck, but the false sharing still limited the overall speedup. Aligning and padding the shared data structure improved performance considerably, but made the load imbalance into the limiting bottleneck. A simple static load balancing scheme improved performance slightly. Although coherence was no longer a serious performance bottleneck, we tried adding prefetching (version 3c), which did not significantly improve performance.

To see whether performance optimizations can carry over to different platforms, we also ported version 3a of the application back to SGI Power Challenge using the shared-memory programming model. The new version outperforms the old Fortran version by 43% (see Figure 12). More detailed experiments showed that the performance gain is mainly attributable to eliminating the serial bottleneck. Padding and aligning only contribute up to 3% performance improvement. Since the SGI Power Challenge is an SMP with a much faster network, it is less sensitive to false sharing. The speedup curves in Figure 12 show that the best speedup on COW is slightly better than the speedup on the SGI Power Challenge. This result is encouraging. It indicates that with the help of proper performance measurement tools and performance tuning, a distributed shared-memory machine can perform as well as a hardware shared-memory machine.

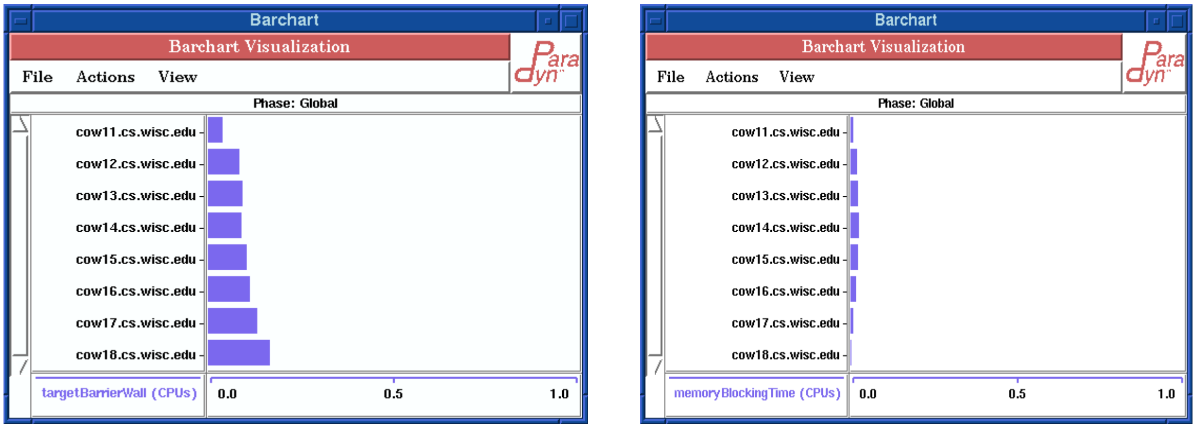

Figure 10: Barrier Time Shows Load Imbalance (version 3a)

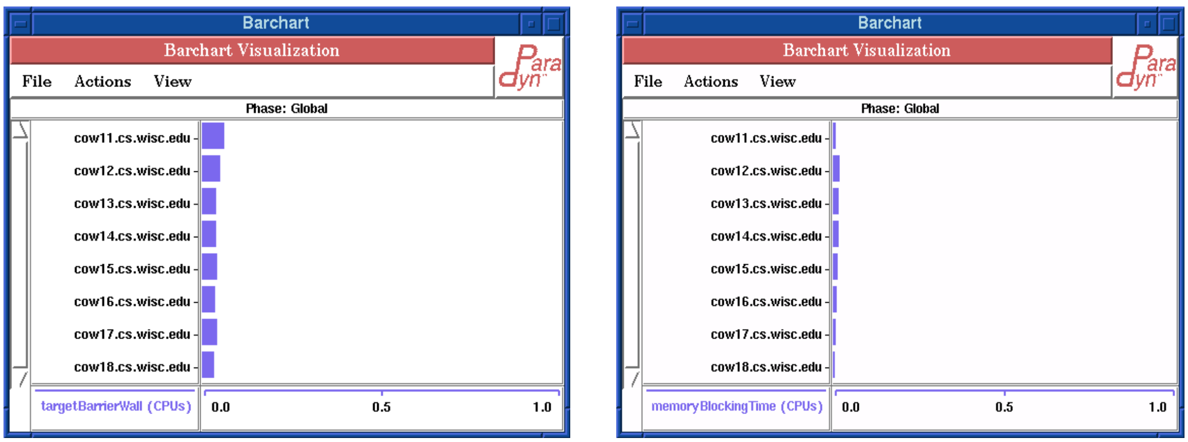

Figure 11: Barrier and Memory Blocking Time after Static Load Balance (version 3b)

Other people have also used the Blizzard version of Paradyn to find shared-memory performance bottlenecks. Brian Toonen used the tool to search for and validate performance bottlenecks in the DSM system itself [28]. Satish Chandra used the tool to monitor the performance of the custom protocols he wrote for his HPF compiler for Blizzard [3]. Trishul Chilimbi used the tool to tune the performance of a database storage management system.

4.5 Discussion

The memory performance tool successful found the performance bottlenecks in this application. Memory profiling helped isolate the shared data structure that incurred the most coherence misses. Pattern detection helped explicate the algorithm's sharing pattern and isolate the false sharing. However, memory profiling by itself would not have found all problems. Paradyn's other performance measurement facilities helped find the sequential bottlenecks, load imbalance, and excessive barrier time.

Custom protocols offer a powerful, but time-consuming option for improving program performance. One unanticipated benefit of memory profiling is that it can indicate when custom protocols are not necessary to improve performance.

5 Automated Search for Bottlenecks

Automating the search for bottlenecks can make performance tuning less difficult. Paradyn's Performance Consultant searches for performance bottlenecks in different foci by testing prewritten hypotheses. A hypothesis specifies a potential performance bottleneck in terms of performance metrics. It is usually a simple function of performance metrics and thresholds. For example, the hypothesis `CPUBound` says there is a performance bottleneck if the ratio of the two performance metrics `cpu_time` and `wall_time` exceeds an adjustable threshold. Hypotheses are organized into a tree-structured hierarchy. The Performance Consultant guides the search with heuristics that attempt to reduce search overhead and find bottlenecks quickly

To automate the search for shared-memory performance bottlenecks, we added new hypotheses for the shared-memory metrics and integrated them into the exist-

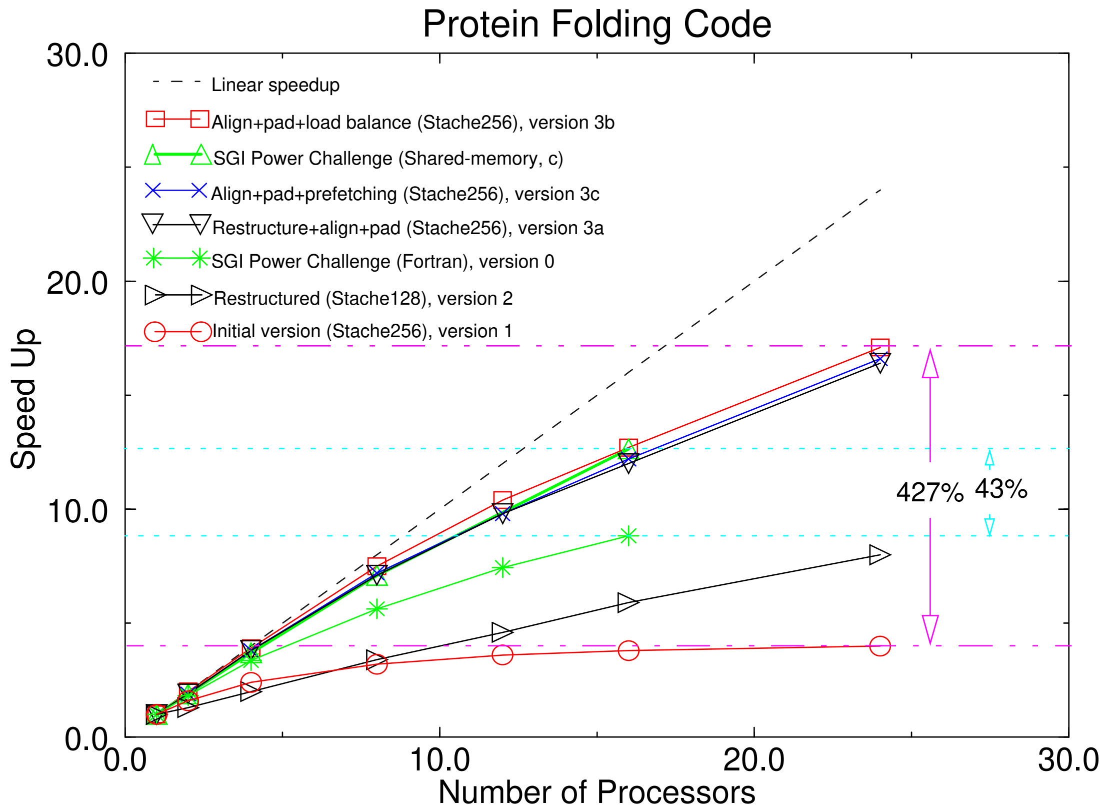

Figure 12: Performance Tuning Result

ing hierarchy. Figure 13 shows the new hypothesis hierarchy, rooted at "Memory". Memory bottlenecks are divided into read miss and write miss. Write is further subdivided into misses due to writes to read-only blocks and misses due to writes to invalid blocks. All hypotheses then subdivide into misses at home and non-home nodes.

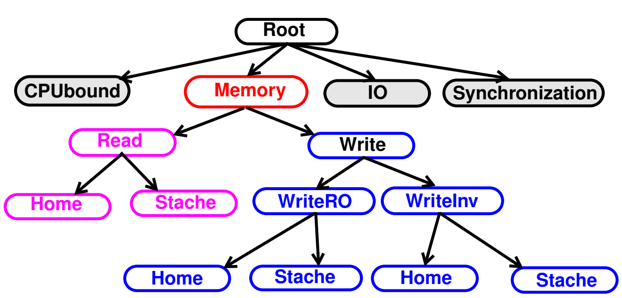

Figure 13: Performance Bottleneck Hypothesis Hierarchy, with Memory Hiearchy Expanded

As a demonstration, we applied the Performance Consultant to version 2 of the program. The search results are shown in Figure 14. The blue (dark gray, in black&white) nodes are problems that the Performance Consultant successfully identified. The first is an "ExcessiveMemoryBlockingTime" bottleneck, which the Performance Consultant further refined to individual data structures. Parts of four particle data structures (`GM->part_0_` to `GM->part_4_`) caused enough memory blocking time to be identified as causes. Note also that two modules (`renumber.c` and `fold4.c`) were identified as bottlenecks, which provided both data and code-centric views of the performance problem.

To make automated search more effective, we are extending the Performance Consultant to make it more configurable, so that we can specify search order, and search heuristics according to the programming model used.

6 Related Work

In addition to the systems cited previously, several profiling tools address shared memory programs.

StormWatch [4] is tool that visualizes coherence protocol actions and links it to a program's behavior. StormWatch provides three linked, graphic views: trace, communication, and source, which reflect the multiple levels of abstraction in an application's behavior. There are two major differences between our work and StormWatch. First, Storm Watch is a postmortem tool that analyzes traces. Our mechanisms use dynamic instrumentation, which is less costly and more scalable. Second, StormWatch provides a more detailed, protocol-specific view of a program's execution,

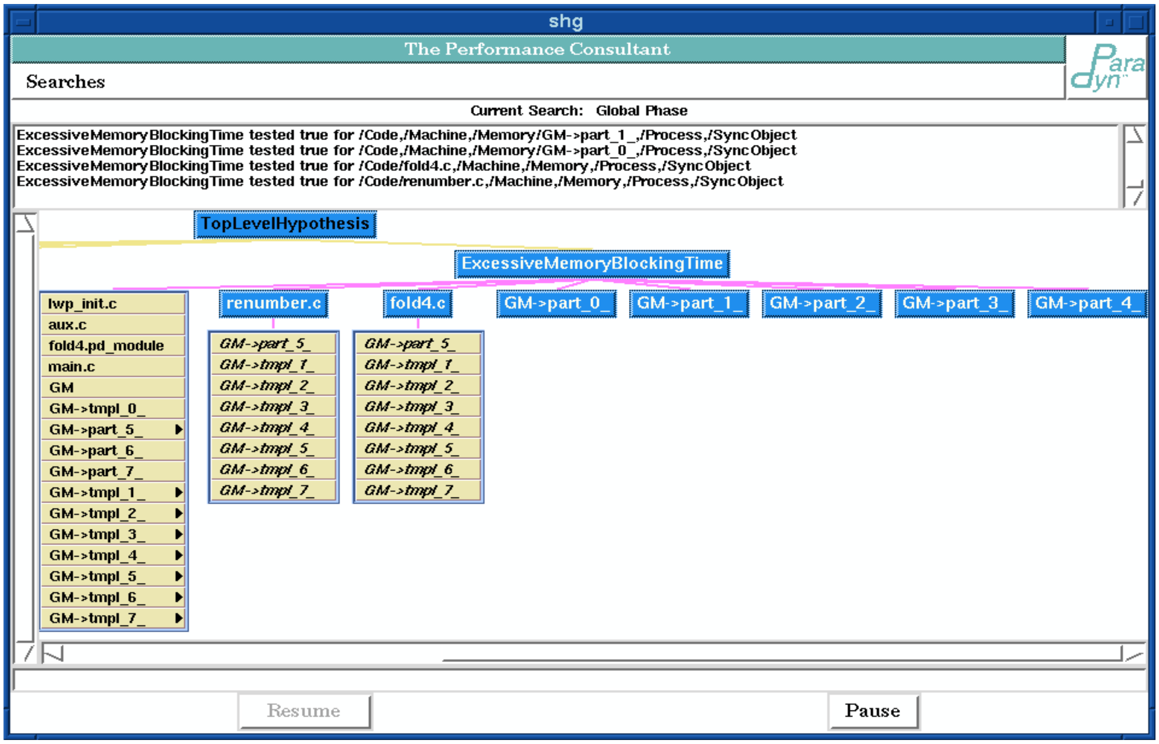

Figure 14: Results of Running the Performance Consultant (version 2)

The blue (darker boxes in black&white) shows the bottlenecks that were identified by the Performance Consultant. "ExcessiveMemoryBlockingTime" is an overall bottleneck, and was further refined to individual data structures (`GM->part_0_`*,* `GM->part_1_`*,* `GM->part_2_`*,* `GM->part_3_` *and* `GM->part_4_`*) and modules (*`renumber.c` *and* `fold4.c`*). The yellow (light grey in b&w) nodes show false search nodes.*

while Paradyn is a general purpose performance tool.

Martonosi et al observed that monitoring memory performance in a multiprocessor is similar to enforcing cache coherence [21]. They have also noticed the importance of categorizing miss counts according to the code and data structures that incurred the misses. Their work focused on exploiting unused issue slots in protocol processors to monitor performance. We dynamically instrument the protocol code. In addition, their monitoring mechanisms only collected memory statistics, such as cache misses and latencies, and did not detect patterns.

Others have extended cache-coherence protocols to detect and optimize migratory sharing behavior. Cox et al evaluated adaptive protocols for bus-based and directory-based systems [5]. Stenstrom et al presented an adaptive protocol to take advantage of migratory behavior of a shared-memory application [26]. Our pattern detection serves a different role—to detect particular access patterns, but not optimize their communication—which makes our implementation simpler and cheaper. Moreover, our mechanisms detect patterns other than migratory behavior.

7 Conclusion

This paper describes a new approach to shared-memory performance profiling. This approach consists of detecting sharing patterns and data-centric presentation of the memory performance data. Pattern detection reports memory access patterns that indicate the possibility of excessive communication. Our pattern detection is implemented as an augmented coherence protocol, so it incurs little additional overhead. Although our current implementation is Blizzard-specific, the approach of detecting sharing pattern by observing coherence events is applicable to many systems.

Memory profiling uses a data-centric view of performance to help a programmer find and understand the performance problems with shared-memory accesses. To organize a large volume of information, we present the shared-memory resources in a two-level hierarchy: data structure and cache blocks belong to a data structure. The data structure view helps relate performance problems to high-level programming language constructs. The cache block view helps isolate specific problems, such as false sharing. Moreover, integration of memory profiling with Paradyn's existing control-oriented view provides a complete view of a shared-memory program's performance. This approach is feasible because Paradyn's dynamic instrumentation reduces the overhead of monitoring by installing instrumentation only when and where it is necessary.

As a test case, we applied these mechanisms to identify and eliminate performance problems in a shared-memory application. With Paradyn, we improved the application's performance by more than a factor of four. The case study demonstrates how these mechanisms can help identify shared-memory performance problems. In addition, the example also shows that, with the help of proper performance measurement tool to optimize communication, a distributed shared-memory machine can be as scalable as a hardware shared-memory system.

ACKNOWLEDGMENTS

Thanks to Sang Tae Kim and Atipat Rojnuckarin for providing the application code and the insight and effort for tuning its performance. Ari Tamches, Mark Callaghan, and Guhan Viswanathan provided insightful discussions. Ari Tamches, Marcelo Gonçalves, Karen Karavanic, Tia Newhall, Oscar Naim and Ling Zheng helped understand Paradyn, Ioannis Schoinas and Babak Falsafi helped with Blizzard, and Satish Chandra helped with Teapot. Mark Hill suggested using software DSM on hardware-only systems. Thanks to Trishul Chilimbi, Brian Toonen, and Satish Chandra for feedback about this tool.

REFERENCES

[1] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *13th ACM Symp. on Operating Systems Principles,* Oct. 1991.

[2] S. Chandra, B. Richards and J. R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. *SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[3] S. Chandra and J. R. Larus. Optimizing Communication in HPF Programs for Fine-Grain Distributed Memory. *6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. Alexis Park Resort, Las Vegas, Nevada, June 18-21, 1997.

[4] T. M. Chilimbi, T. Ball, S. G. Eric, J. R. Larus. StormWatch: A Tool for Visualizing Memory System Protocols. *Supercomputing'95*, San Diego, CA, December, 1995.

[5] A. L. Cox and R. J. Fowler. Adaptive Cache coherency for Detecting Migratory Shared Data. *20th Annual Int'l Symp. on Computer Architecture,* May 1993.

[6] F. Dahlgren, M. Dubois, and P. Stenstrom. Combined Performance Gains of Simple Cache Protocol Extensions. *21th Annual Int'l Symp. on Computer Architecture*, April 1994.

[7] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for user-level Shared Memory. *Supercomputing'94*, November, 1994.

[8] A. Gupta, M. Martonosi, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. *Performance Evaluation Review* **20**, 1, June 1992.

[9] M. D. Hill, J.R. Larus, S.K. Reinhardt, and D.A. Wood. Tempest: A Substrate for Portable Parallel Programs. *COMPCON*'95, San Francisco, March 1995.

[10] J.K. Hollingsworth and B.P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", *Int'l Conf. on Supercomputing*, Tokyo, July 1993.

[11] J.K. Hollingsworth, B. P. Miller, M. J. R. Gonçalves, O. Naim. Z. Xu and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. Tech.l Report, Comp. Science Department, UW-Madison.

[12] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *23rd Annual Int'l Symp. on Comp. Architecture*, Philadelphia PA, May 1996.

[13] G. A. Huber and S. Kim. Weighted-Ensemble Brownain Dynamics Simulations for Protein Association Reactions. *Biophysical Journal*, Vol. 70, January 1996.

[14] R.B. Irvin and B.P. Miller, "A Performance Tool for High-Level Parallel Programming Languages" in **Programming Environments for Massively Parallel Distributed Systems,** Birkaeuser Verlag, Basel, K.M. Decker and R.M. Rehmann, eds., 1994.

[15] R.B. Irvin and B.P. Miller, "Mapping Performance Data for High-Level and Data Views of Parallel Program Performance", *Int'l Conf. on Supercomputing*, Philadelphia, May 1996.

[16] K. J. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High Performance All-Software Distributed Shared Memory. *15th ACM Symp. on Operating System Principles (SOSP),* Copper Mountain, Colorado, December 1995.

[17] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *IEEE Computer* **29**, 2, February 1996.

[18] J. Kuskin et al. The Stanford FLASH Multiprocessor. 2*1st Annual Int'l Symp. on Comp. Architecture*, April 1994.

[19] A. R. Lebeck and D. A. Wood. Cache profiling and spec benchmarks: A case study. *IEEE Computer* **27**, 10, October 1994.

[20] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. *23th Annual Int'l Symp. on Comp. Architecture*, Philadelphia PA, May 1996.

[21] M. Martonosi, D. Ofelt and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. *ACM Sigmetrics Conf. on Measurement & Modeling of Comp. Systems*, Philadelphia, PA, May, 1996.

[22] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and Tia Newhall. The Paradyn Performance Tools. *IEEE Computer* **28**, 11, November 1995.

[23] S. K. Reinhardt, J. R. Larus, D. A. Wood. Typhoon and Tempest: User-Level Shared Memory. *21st Int'l Symp. on Comp. Architecture*, April 1994.

[24] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. *8th Int'l Conf. on Architectural Support for Programming Languages and Operating Sys. (ASPLOS), 1996*.

[25] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, D. A. Wood. Fine-grained Access Control for Distributed Shared Memory. In *Pr 6th Int'l Conf. on Architectural Support for Prog. Languages and Operating Sys. (ASPLOS),* Oct. 1994.

[26] P. Stenstrom, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol for Optimized Migratory Sharing. *20th Annual Int'l Symp. on Comp. Architecture*, May 1993.

[27] Sun Microelectronics. *UltraSPARC User's Manual*. 1996.

[28] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Shoinas, M.D. Hill and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. *6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. Las Vegas, June 1997.