Evaluation, Implementation and Performance of Write Permission Caching in the DSZOOM System

Håkan Zeffer, Zoran Radović, Oskar Grenholm and Erik Hagersten

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2004-005
February (updated June) 2004
ISSN 1404-3203

EVALUATION, IMPLEMENTATION AND PERFORMANCE OF WRITE PERMISSION CACHING IN THE DSZOOM SYSTEM

Håkan Zeffer, Zoran Radović, Oskar Grenholm and Erik Hagersten

Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
http://www.it.uu.se/research/group/uart

ABSTRACT

Fine-grained software-based distributed shared memory (SW-DSM) systems typically maintain coherence with in-line checking code at load and store operations to shared memory. The instrumentation overhead of this added checking code can be severe.

This paper (1) shows that most of the instrumentation overhead in the fine-grained DSZOOM SW-DSM system is store related, (2) introduces a new write permission cache (WPC) technique that exploits spatial store locality and batches coherence actions at runtime, (3) evaluates WPC and (4) presents WPC results when implemented in a real SW-DSM system. On average, the WPC reduces the store instrumentation overhead in DSZOOM with 42 (67) percent for benchmarks compiled with maximum (minimum) compiler optimizations.

1. INTRODUCTION

The idea of implementing a shared address space in software across clusters of workstations, blades or servers was first proposed almost two decades ago [1]. The most common approach, often called shared virtual memory (SVM) [1, 2], uses the virtual memory system to maintain coherence. An SVM system provides a coherent shared virtual address space at page granularity, using the local page table for access control and message passing for inter-node communication [2]. Unfortunately, the large coherence unit size used in SVM systems introduces false sharing. To increase performance, SVM systems often use loose memory consistency models and multiple writer protocols [3, 4, 5, 6, 7, 8, 9].

Another popular approach is fine-grain systems, which maintain coherence by instrumenting memory operations in the programs [10, 11, 12]. An advantage with these systems is that they avoid the high degree of false sharing common in SVMs. Hence, they can implement stricter memory consistency models and run applications written for hardware-coherent multiprocessors. However, fine grain systems suffer from relatively high instrumentation overhead [10, 11, 12]. Multiple schemes to reduce this overhead have been proposed. For example, Shasta [11] statically merges coherence actions at instrumentation time.

DSZOOM [12] is a sequentially consistent [13], fine-grained software distributed shared-memory (SW-DSM) system. The core of the DSZOOM system depends on binary/assembler instrumentation. That is, the instrumentation tool inserts access control checks (and write permission checks) after/before loads/stores to shared memory. A unique aspect of DSZOOM is that it completely removes the interrupt- and/or poll-based asynchronous protocol messaging that traditional SW-DSM systems suffer from.

In this paper, we show that most of the instrumentation overhead in DSZOOM comes from store instrumentation. We propose a dynamic *write permission cache* (WPC) technique that exploits spatial store locality. This technique dynamically merges coherence actions at runtime. We evaluate the proposed WPC technique in a parallel simulator. In addition, we present and evaluate two real WPC implementations as part of the DSZOOM system.

The WPC reduces DSZOOM's average store instrumentation overhead with 42 percent for applications compiled with the highest optimization level and 67 percent for non-optimized SPLASH-2 benchmarks. The parallel execution time for 16-processor runs for a 2-node configuration is reduced as much as 27 (32) percent and on average 7 (11) percent for benchmarks compiled with maximum (minimum) compiler optimization.

The rest of this paper is organized as follows: Section 2 gives a brief background of DSZOOM and how it differs from most other SW-DSM systems. In Section 3 and 4, we introduce and evaluate the new write permission caching technique. The real WPC implementation, instrumentation overhead and parallel performance are presented in Section 5. A discussion about WPC related deadlocks and memory consistency issues is given in Section 6. Finally, we discuss related work and conclude.

2. DSZOOM BACKGROUND

This Section contains an overview of the general DSZOOM system [12]. DSZOOM is a sequentially consistent [13], fine-grained SW-DSM system that differs from most other SW-DSM implementations in five major areas:

1. *Fine-grain coherence* : Instead of relying on page-based coherence units, the *code instrumentation* [14, 15] technique is used to expand load and store instructions that may touch shared data into a sequence of instructions that performs in-line coherence checks. This work is originally inspired by DEC's Shasta [11, 16, 17] and Wisconsin's Blizzard [10, 18] proposals from the mid 1990s.

2. *Protocol in requesting processor* : If a coherence check determines that a global coherence action is required, the entire protocol is run in the requesting processor, which otherwise would have been idle waiting for the coherence action to be resolved.

3. *No asynchronous interrupts* : Since the entire coherence protocol is run in the requesting processor/node, there is no need to send asynchronous interrupts to "coherence agents" in other nodes. There is also no need to use polling. I.e., this removes the overhead of asynchronous interrupts from the remote latency-timing path.

4. *Blocking directory coherence protocol* : The global coherence protocol is designed to avoid all traditional corner cases of cache coherence protocols. Only one thread at a time can have the exclusive right to produce global coherence activity at each piece of data. A blocking protocol simplifies the task of designing a correct and race free protocol.

5. *Thread-safe protocol implementation* : While most other implementations have only one protocol thread per node, our scheme allows several threads in the same node to perform protocol actions at the same time.

DSZOOM assumes a high-bandwidth, low-latency cluster interconnect, supporting fast user-level mechanisms for *put*, *get* and *atomic* operations to remote nodes' memories. We further assume that the write order between any two end-points in the network is preserved. Early interconnects [19, 20, 21] does not support atomic operations. However, emerging interconnects, such as InfiniBand [22], does.

Each DSZOOM node consists of a symmetric multiprocessor (SMP) or hardware DSM (HW-DSM) multiprocessor. The SMP or HW-DSM hardware keeps coherence among the caches and the memory within each node. The InfiniBand-like interconnect, as described above, connects the nodes.

The DSZOOM technology could be applicable to a wide range of cluster implementations, even though this specific implementation is targeting SPARC-based Solaris systems.

2.1. Blocking Directory Protocol

Most of the complexity of a coherence protocol relates to the race conditions caused by multiple simultaneous requests for the same coherence unit. Blocking directory coherence protocols have been suggested to simplify the design and verification of HW-DSM systems [23]. The directory blocks new requests to a coherence unit until all previous coherence activity to that coherence unit has completed. This eliminates all race conditions, since each coherence unit can only be involved in one coherence activity at any specific time.

The DSZOOM protocol is a distributed version of a blocking directory protocol. A processor that has detected the need for global coherence activity will first acquire a global lock associated with the coherence unit before starting the coherence activity. The assumption that atomic operations are provided by the network makes it possible to remove asynchronous interrupts in remote nodes. A requesting processor can lock a remote directory entry and independently obtain read and write permission.

The DSZOOM protocol states, `MODIFIED`, `SHARED` and `INVALID` (MSI), are explicitly represented by global data structures in the nodes' memories. Effective address bits of memory operations determine the location of a coherence unit's directory location, i.e., its "home node." All coherence units in `INVALID` state store by convention a "magic" data value, as independently suggested by Schoinas et. al. [18] and Scales et. al. [11]. This significantly reduces the number of directory accesses caused by load operations, since the directory only has to be consulted if a local load returns the magic value.

To reduce the number of accesses to remote directory entries caused by global store operations, each node has one byte of local state (MTAG) per global coherence unit (similar to the *private state table* found in Shasta [16]), indicating if the coherence unit is locally writable. Before each global store operation, the MTAG byte is checked. The directory only has to be consulted if the MTAG indicates that the node currently does not have write permission to the coherence unit. Since a home node can detect if it has write permission with local memory references (the directory is located in its local memory) home nodes does not need any extra MTAG state.

To avoid race conditions, all directory and MTAG entries have to be locked before a write permission check is carried out. Otherwise, a coherence unit can be downgraded between the consultation and the point in time where

```
1: original_store:
2:     ST Rx, addr;
```

Fig. 1. Ordinary store.

```
1: original_store_snippet:
2:     LOCK(MTAG_lock[addr]);
3:     LD Ry, MTAG_value[addr];
4:     if (Ry != WRITE_PERMISSION)
5:         call st_protocol;
6:     ST Rx, addr;
7:     UNLOCK(MTAG_lock[addr]);
```

Fig. 2. Ordinary store snippet.

the store is performed. The content of the directory/MTAG entry may reveal the need to call the store coherence protocol in order to obtain write permission. When the processor has write permission, the store can be performed and the directory/MTAG released.

3. WRITE PERMISSION CACHE (WPC)

The applications studied have more loads than stores to shared memory. Yet, the store-related coherence checks stand for the largest part of the instrumentation overhead, see Section 5.3. Most of this overhead comes from the fact that we have to lock the directory and/or MTAG entries before the write permission check is performed.

The idea with the *write permission cache* (WPC) is to minimize locking and consulting/checking of MTAGs by exploiting spatial store locality. Instead of releasing the local directory/MTAG lock after a store is performed, a thread holds on to the write permission and the lock, hoping that the next store will be to the same coherence unit. If indeed the next store is to the same coherence unit, the store overhead is reduced to a few ALU operations and a conditional branch instruction. Only when a WPC miss occurs, a lock release and a new lock acquire has to be performed. That is, when a store to another coherence unit appears.

Figures 1 and 2 show how an ordinary store instruction (Figure 1) expands to a *store snippet* (Figure 2) when the original DSZOOM system is used. `Ry` is a temporary register, `Rx` contains the value to be stored and `addr` is the effective address of this particular store operation. On lines 2 and 7 in Figure 2, the local directory/MTAG lock is acquired and released. Lines 3 and 4 load and check the directory/MTAG value for permission. If the processor does not have write permission, the store protocol is called at line 5. Finally, at line 6, the local store is performed.

Figure 3 shows how a WPC optimized store snippet is

```
1: wpc_fast_path_snippet:
2:     if (WPC != CU_id[addr])
3:         call slow_path
4:     ST Rx, addr
5:
6: wpc_slow_path_snippet:
7:     UNLOCK(MTAG_lock[WPC]);
8:     WPC = CU_id[addr];
9:     LOCK(MTAG_lock[addr]);
10:    LD Ry, MTAG_value[addr];
11:    if (Ry != WRITE_PERMISSION)
12:        call st_protocol;
```

Fig. 3. WPC store snippet.

designed. The snippet consists of two parts: a fast- and a slow path. Line 2 checks if the current coherence unit is the same as the one cached in the WPC (a WPC entry contains a coherence unit identifier, refered to as `CU_id[addr]` in Figure 3). If that is the case, then the processor has write permission and can continue its execution. The slow-path code is entered only if a WPC miss occurs. In that case, the processor actually checks for write permission in the local directory/MTAG structure. The slow path in Figure 3 has much in common with the ordinary store snippet found in Figure 2. However, at line 7, the old lock, whose coherence unit identifier is cached in the WPC, has to be released. Moreover, at the end of the snippet, the processor keeps the lock. At line 8, the processor inserts the new coherence unit identifier in its WPC. Memory mappings are created in such a way that the `CU_id[addr]` reference at lines 2 and 8 easily can be done with arithmetic instructions, i.e., a shift. Thus, the fast path contains no extra memory references since thread-private registers are used as WPC entries. In other words, an *n*-entry WPC system with *t* threads contains $n \times t$ WPC entries in total. As in Figure 2, `Ry` is a temporary register, `Rx` contains the value to be stored and `addr` is the effective address of the original store.

4. EVALUATING WPC

This Section evaluates the WPC technique. It presents benchmarks and the simulation environment. It also presents the WPC hit rate numbers and WPC's impact on blocking directory protocol collisions.

4.1. Applications

The benchmarks that are used in this paper are well-known workloads from the SPLASH-2 benchmark suite [24]. The data set sizes for the applications studied are presented in Table 1. The applications are representative for scientific,

Program	Problem Size
barnes	16k particles
cholesky	tk29.O
fft	1M points
fmm	32k particles
lu-c	1024×1024 matrices, 16×16 blocks
lu-nc	1024×1024 matrices, 16×16 blocks
ocean-c	514×514
ocean-nc	258×258
radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
radix	4M integers, radix 1024
raytrace	car
water-nsq	2197 molecules, 2 iterations
water-sp	2197 molecules, 2 iterations

Table 1. SPLASH-2 benchmark suite.

engineering, and computing graphic fields.

The reason why we cannot run `volrend` is that shared variables are not correctly allocated with the *G_MALLOC* macro.

4.2. Simulation Environment

We have developed a simulation environment to be able to test new protocol optimizations such as the WPC. The simulator is called *protocol analyzer* and is designed for rapid prototyping and simulation of realistic workloads on parallel machines.

Our system has much in common with the Wisconsin Windtunnel II [25] simulator. It uses *direct execution* and *parallel simulation* to gain performance. With direct execution, a program from the target system runs on an existing host system. That is, instructions are executed on the host instead of interpreted by the simulator [26]. Parallel simulation exploits a host computer's parallel resources such as large memory and multiple processors to speedup the simulation [25]. These two techniques make simulations with realistic workloads and protocol handling feasible.

The output from instrumented load and store operations of the studied benchmarks is used as a front end to protocol analyzer. Moreover, an SMP is used as host system during simulations. The SMP hardware guarantees coherence, memory consistency and correctness during the parallel execution of the program.

Protocol analyzer implements a configuration system that makes it possible to use different kind of back ends. A protocol analyzer back end can simulate caches, cache coherence protocols and much more by using shared memory and simple counters. However, it is not possible to simulate target system's execution time.

Instrumentation overhead and calls to protocol analyzer back ends can introduce timing errors or skewness in the simulation. It is important to consider these timing issues when analyzing data produced by the simulator.

4.3. Simulated WPC Hit Rate

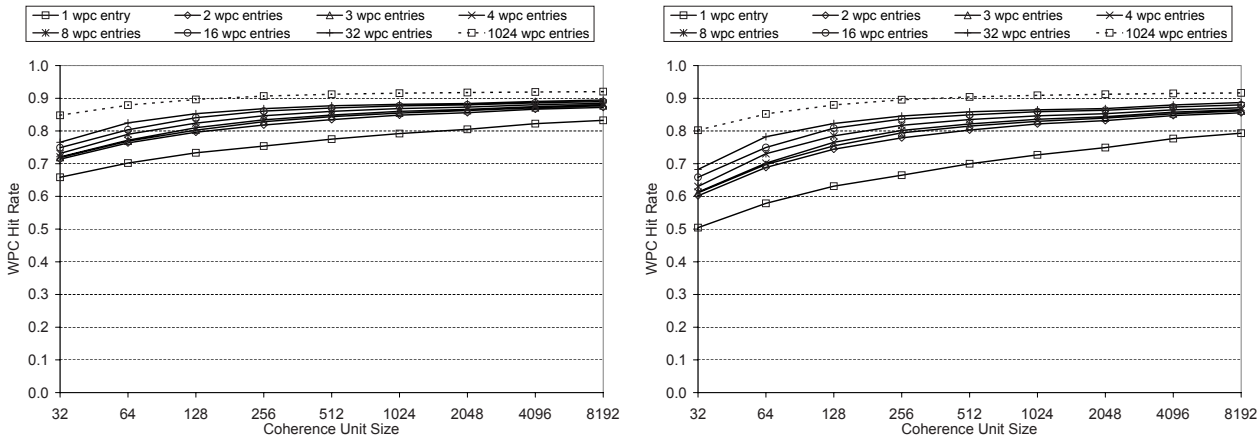
In this Section, we investigate if it is possible to achieve high WPC hit rate with a few WPC entries. This is especially important for an efficient software WPC implementation. We use a multiple-entry WPC model to simulate WPC hit rate in protocol analyzer. WPC hit rate is measured as hits in the WPC divided by the number of stores to shared memory. All data are collected during the parallel execution phase of the application when run with 16 processors. Because each processor has its own set of WPC entries, and each processor simulates its own WPC hit rate, a timing skewing introduced by protocol analyzer is not a problem. This is especially true for the applications that only uses synchronization primitives visible to the runtime system. Moreover, our simulated 1-entry WPC hit-rate numbers have been verified with a slightly modified DSZOOM implementation. The numbers are almost identical (maximum difference is less than 0.03 percent).

Figure 4 shows hit rate for 1, 2, 3, 4, 8, 16, 32 and 1024 WPC entries when coherence unit size is varied from 32 to 8192 bytes. Figure 4 contains average data collected from thirteen SPLASH-2 applications run with 16 processors. Data for individual applications can be found in Appendix A.

Figure 5 shows hit rate for thirteen applications when 1, 2, 3, 4, 8, 16, 32 and 1024 WPC entries and a coherence unit size of 64 bytes is used. Figure 5 contains WPC hit rate for applications compiled with (a) minimum and (b) maximum optimization levels. Applications compiled with a low optimization level seem to have higher WPC hit rates than fully optimized binaries. Still, almost all applications compiled with maximum optimization have a WPC hit rate above 0.7. In particular, this is true when two or more WPC entries are used. If the number of WPC entries is increased from one to two, applications such as `barnes`, `cholesky` and `fft` significantly improve their hit rate numbers. This is due to multiple simultaneous write streams. Increasing the number of entries from two to three or from three to four does not give such a large WPC hit rate improvement. Thus, increasing the number of WPC entries above two might not be justified. `radix` and `raytrace` show poor WPC hit rate. WPC hit rate numbers for other coherence unit sizes (32-8192 bytes) and individual applications are given in Appendix A.

4.4. WPC Impact on Directory Collisions

Data sharing, such as multiple simultaneous requests to directory entries, might lead to processor stall time in blocking



(a) WPC hit rate with minimum compiler optimization.

(b) WPC hit rate with maximum compiler optimization.

Fig. 4. Average WPC hit rate for thirteen SPLASH-2 benchmarks while varying the coherence unit size.

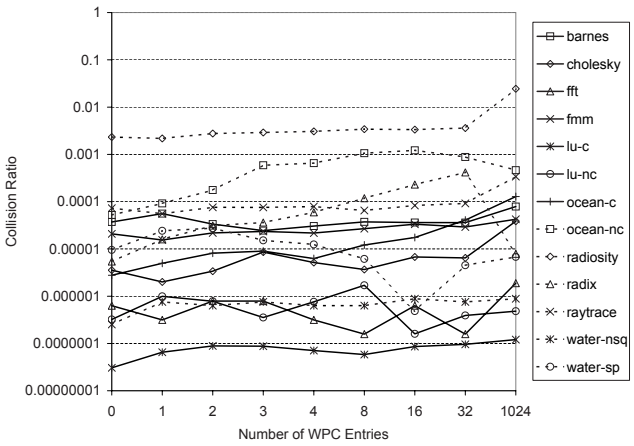


Fig. 6. WPC impact on directory collisions.

protocols. Consider a case where two processors simultaneously try to obtain write permission for a coherence unit. Since a processor always has to lock a directory entry before it can obtain permission, one will have to wait for the other to finish its directory activity before it can continue. We call this a *directory collision*. We believe that it is important to investigate if the number of directory collisions increases when the number of WPC entries are increased.

We have simulated the DSZOOM protocol in protocol analyzer to estimate what impact a WPC has on the number of directory collisions. Figure 6 shows the number of directory collisions divided by the number of locks taken when no WPC entries are used. The number of collisions does not increase significantly when the number of WPC entries is increased. In some cases, the number of directory collisions actually decreases when the number of simulated WPC entries is increased. This is counter intuitive, but indicates that the number of directory collisions are highly timing dependent.

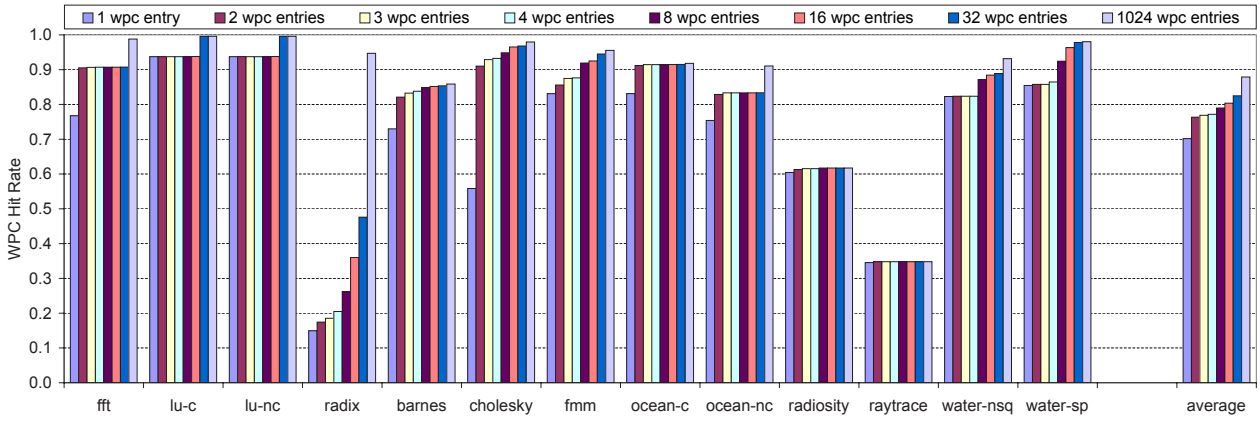
Protocol analyzer might introduce timing skewness during a simulation. However, because memory operations take longer time when protocol analyzer is used, we believe that our simulations are unnecessarily negative. That is why we believe that the number of directory collisions will be even less when run in DSZOOM than the simulation results currently show.

5. SOFTWARE DSM WPC IMPLEMENTATION AND PERFORMANCE

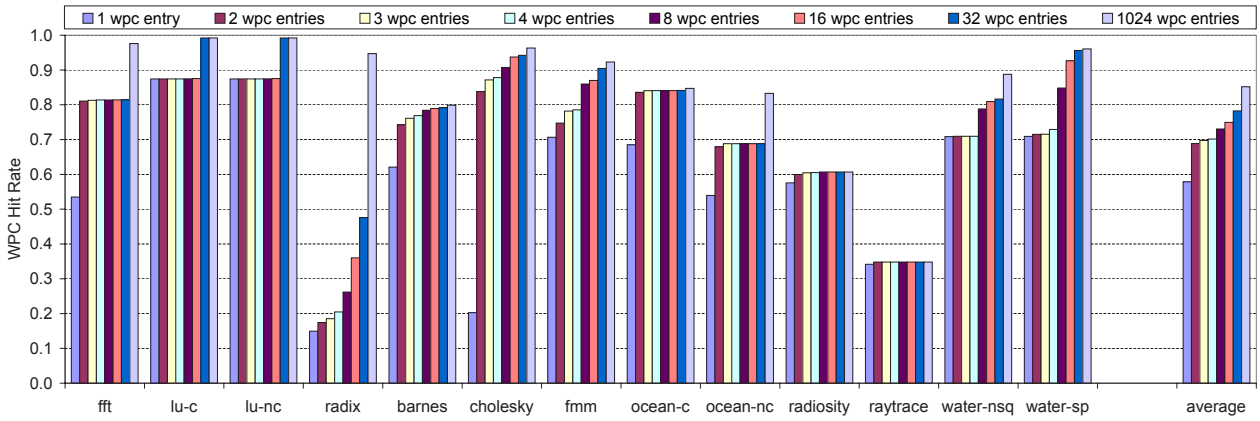
This Section present WPC results run on a real SW-DSM system. In Section 5.1, the experimental setup, such as hardware and compiler, is described. The WPC implementation used in the SW-DSM system is briefly discussed in Section 5.2. Instrumentation overhead and parallel performance are discussed in Section 5.3 and 5.4 respectively.

5.1. Experimental Setup

All sequential experiments in this paper are measured on a Sun Enterprise E6000 server [27]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [28]) and a total bandwidth of 2.7 Gbyte/s. Each processor has a 16 kbyte on-chip instruction cache, a 16 kbyte



(a) WPC hit rate for benchmarks compiled with minimum compiler optimization.



(b) WPC hit rate for benchmarks compiled with maximum compiler optimization.

Fig. 5. WPC hit rate for thirteen SPLASH-2 benchmarks using a coherence unit size of 64 bytes.

on-chip data cache, and a 4 Mbyte second-level off-chip data cache.

The HW-DSM numbers have been measured on a 2-node Sun WildFire built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [23, 29]. The WildFire system has been configured as a traditional cache-coherent, non-uniform memory access (CC-NUMA) architecture with its data migration capability activated while its coherent memory replication (CMR) has been kept inactive. The Sun WildFire access time to local memory is the same as above, 330 ns, while accessing data located in the other E6000 node takes about 1700 ns (lmbench latency). The E6000 and the WildFire DSM system are both running a slightly modified version of the Solaris 2.6 operating system.

Both the original DSZOOM and DSZOOM with a WPC, from now on called DSZOOM-base and DSZOOM-WPC respectively, run in user space on the Sun WildFire system. The WildFire interconnect is used as a cluster interconnect between the two DSZOOM nodes. Block load, block store and atomic memory operations are used as remote put, get and atomic operations. The DSZOOM system guarantees the coherence of the applications, that is, the hardware coherence is not used. Moreover, the data migration and the CMR data replication of the WildFire interconnect are inactive when DSZOOM runs.

All experiments in this paper use GCC 3.3.3 and a simple custom-made assembler instrumentation tool for UltraSPARC targets. To simplify instrumentation, we use GCC's `-fno-delayed-branch` flag that avoids loads and sto-

res in delay slots. We also use the `-mno-app-regs` flag that reserves UltraSPARC's thread private registers for our snippets. These two flags slow down SPLASH-2 applications with less than 3 percent (avg.). Compiler optimization levels are `-O0` (earlier refered to as minimum compiler optimization) and `-O3` (earlier refered to as maximum compiler optimization). The major limitation with this approach is that the source code must be available, which is sometimes not the case for the system libraries or other commercial software components that might access shared data.

5.2. The DSZOOM-WPC implementation

For an efficient DSZOOM-WPC implementation, it is necessary to reserve processor registers for WPC entries to avoid additional memory references in store snippets. However, with multiple WPC entries, the register pressure as well as the WPC checking code increases. As indicated in Section 4.3, a 2-entry WPC may be a good design choice. Thus, in this paper, we implement and evaluate 1- and 2-entry WPC systems.

The WPC technique raises memory consistency model, dead- and livelock concerns: All DSZOOM implementations presented in this paper implement the same memory consistency model (sequential consistency), with or without WPC. A detailed discussion can be found in Section 6.

Most of the dead- and livelock issues are solved by the DSZOOM runtime system. A processor's WPC entries have to be released at (1) synchronization points, at (2) failures to acquire directory/MTAG entries and at (3) thread termination. However, user level flag synchronization can still introduce WPC related deadlocks. The WPC deadlock problem and three suggested solutions are discussed in Section 6. In this study, applications that use user level flag synchronization (`barnes` and `fmm`) are manually modified with WPC release code.

5.3. Instrumentation Overhead

In this Section, we characterize the overhead of inserted fine-grain access control checks for global loads/stores for all of the studied SPLASH-2 programs. Since the WPC technology is a store optimization technique, the write permission checking code (store snippets) is the focus of this Section. To obtain a sequential instrumentation breakdown for different snippets, we ran the applications with just one processor and with only one kind of memory instruction instrumented at a time. This way, the code will never need to perform any coherency work and will therefore never enter the protocol code (written in C).

Sequential instrumentation overhead breakdown for applications compiled with maximum and minimum compiler optimizations is shown in Figure 7. The store overhead is the single largest source of the total instrumentation overhead: 61 (34) percent for optimized (non-optimized) code. In addition, the single-WPC checking code (*st-swpc*) reduces this store overhead to 57 (16) percent. Double-WPC checking code (*st-dwpc*) further reduces the original store overhead to 36 (11) percent. As expected, the reduction is most significant for `lu-c` and `lu-nc` because they have the highest WPC hit rate, see Figure 5, and low shared load/store ratio [24]. `fft` and `cholesky` perform much better when a 2-entry WPC is used. For `radix`, the instrumentation overhead slightly increases for the *st-swpc* and *st-dwpc* implementations. The low WPC hit rate (see Figure 5) is directly reflected in this particular instrumentation breakdown. Finally, the "perfect" WPC checking code (*st-wpc-hr1*) demonstrates very low instrumentation overheads: 9 percent for optimized and 3 percent for non-optimized code.

5.4. Parallel Performance

In this Section, the parallel performance of two WPC-based DSZOOM systems is studied. Figure 8 shows normalized execution time for Sun Enterprise E6000 (SMP), 2-node Sun WildFire (HW-DSM) and three DSZOOM configurations:[1]

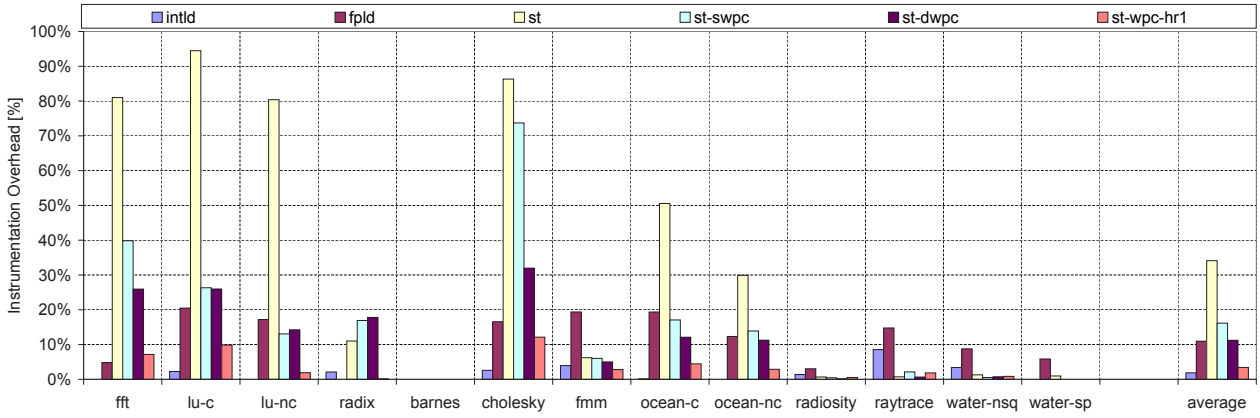
1. DSZOOM-base: the original DSZOOM implementation.
2. DSZOOM-swpc: the DSZOOM implementation with a 1-entry WPC.
3. DSZOOM-dwpc: the DSZOOM implementation with a 2-entry WPC.

All DSZOOM configurations use a coherence unit size of 64 bytes. Both the HW-DSM configuration and the DSZOOM configurations run on two nodes and with eight processors per node (16 in total). The WPC technique improves the parallel DSZOOM performance with 7 (11) percent for benchmarks compiled with maximum (minimum) compiler optimization levels. The performance gap between the hardware-based DSM and the DSZOOM system is reduced with 14 (31) percent. Thus, the DSZOOM slowdown is in the range of 77 (40) percent compared to an expensive hardware implementation of shared memory, both running optimized (non-optimized) applications.

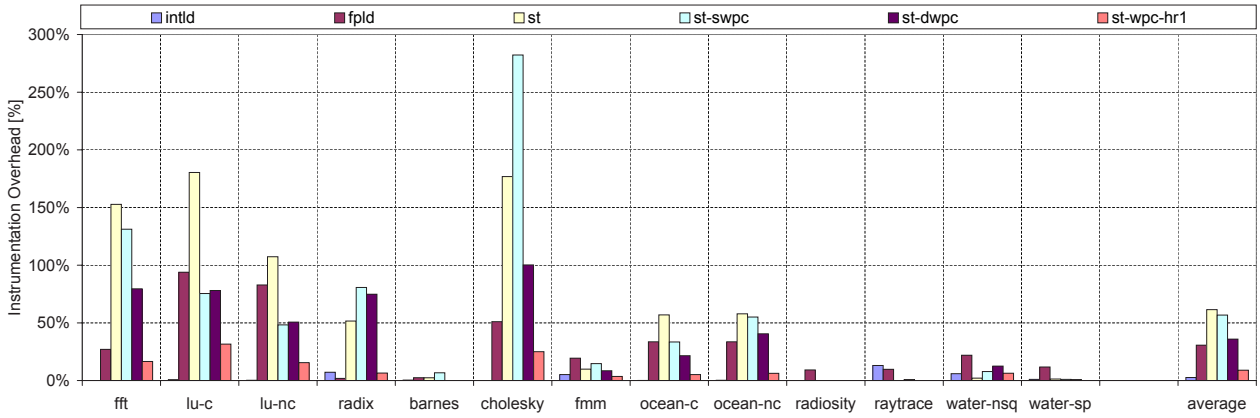
6. DEADLOCK, PROGRAMMING MODEL AND MEMORY CONSISTENCY ISSUES

The protocol of the base architecture maintains sequential consistency [13] by requiring all the acknowledges from the

[1] Real execution times are shown in Appendix C.



(a) Sequential instrumentation overhead breakdown for non-optimized binaries.



(b) Sequential instrumentation overhead breakdown for fully optimized binaries.

Fig. 7. Instrumentation overhead for integer loads (*intld*), floating-point loads (*fpld*), the original store snippet (*st*), a 1-entry WPC store snippet (*st-swpc*), a 2-entry WPC store snippet (*st-dwpc*) and a store snippet with WPC hit rate 1.0 (*st-wpc-hr1*).

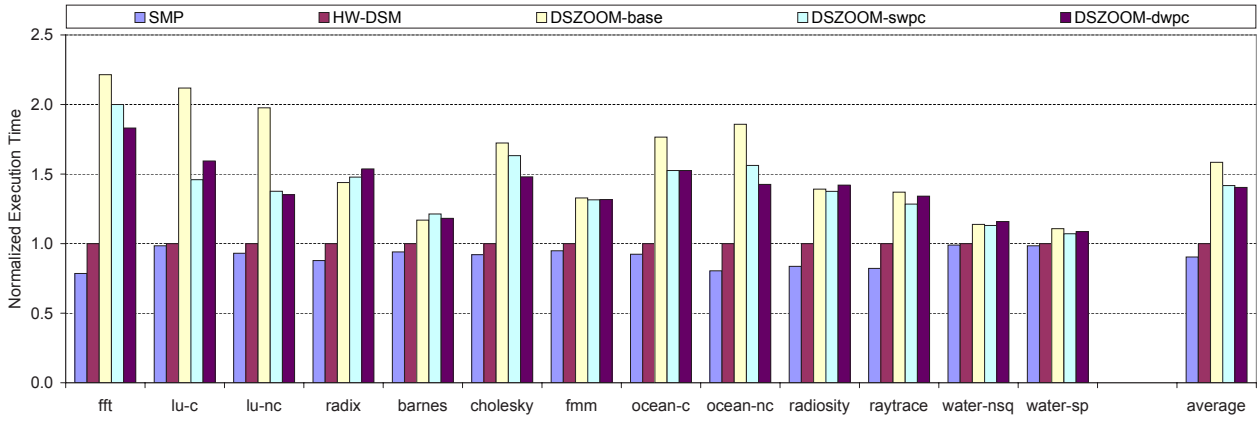
sharing nodes to be received before a global store request is granted. Introducing the WPC will not weaken the memory model. The WPC protocol still requires all the remotely shared copies to be destroyed before granting the write permission. WPC just extends the duration of the permission tenure before the write permission is given up. Of course, if the memory model of each node is weaker than sequential consistency, it will decide the memory model of the system (the system implements total store order (TSO) if multiple SPARC processors are used per node).

The strict memory consistency model allows the usage of user level flag synchronization even though it is not visible to the runtime system. However, the code in Figure 9 can lead to a WPC related deadlock. Let two processors, CPU0

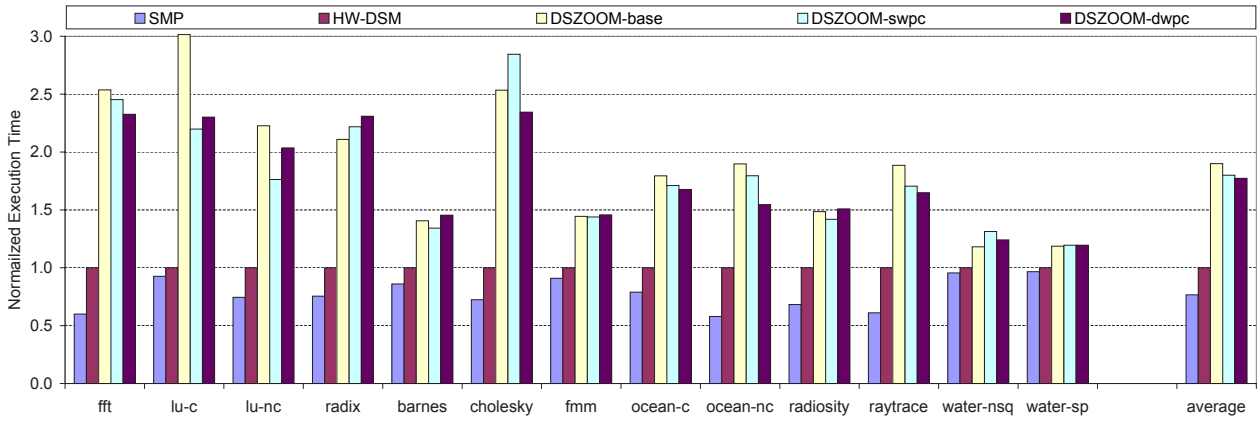
```
01:     /* CPU0's code */
02:     a = 1;
03:     while (flag != 1); /* wait */
04:     ...

11:     /* CPU1's code */
12:     b = 1;
13:     flag = 1;
14:     ...
```

Fig. 9. DSZOOM-WPC deadlock example code.



(a) 16-processor runs and non-optimized binaries.



(b) 16-processor runs and fully optimized binaries.

Fig. 8. Parallel performance for 16-processor configurations.

and CPU1, execute the code shown in Figure 9. CPU0 enters the code first (executes line 02) and assigns the global variable *a* the value one. This implies that CPU0 puts *a*'s coherence unit *u* in its WPC. When CPU0 reaches line 03, it starts to spin on the shared variable *flag*, waiting for CPU1. *flag* is located on another coherence unit than *a* and is initially assigned the value zero. CPU1 enters the code (line 12) and tries to obtain write permission for *b*. However, the global variable *b* is located on the same coherence unit *u* as the global variable *a* (false sharing). Since the directory state for *u* is locked and cached by CPU0's WPC we have a deadlock! CPU0 is waiting for CPU1 to update the *flag* variable, and CPU1 is waiting for CPU0 to release the caching of *u*. This deadlock is resolved if CPU0 flushes its WPC entries.

From the simple example shown in Figure 9, it is clear that DSZOOM-WPC has to implement a deadlock avoidance mechanism if synchronization not visible to the runtime system is to be supported. Here, we briefly discuss three such mechanisms.

- All processors' WPC entries can be flushed periodically based on timer interrupts. `setitimer(2)` and a signal handler can, for example, be used to implement such a mechanism. However, interrupt and signal handling takes time and can slow down the execution of the parallel application. It may also lead to unnecessary WPC flushes.

- CPU1 can detect the deadlock and interrupt CPU0. CPU0 can then flush its WPC entries. Hence, the deadlock is resolved. This approach has both good and

bad sides; it is easy for CPU1 to detect the deadlock since it spins in protocol code trying to grab a coherence-related lock. Nevertheless, interrupting another processor with an asynchronous interrupt introduces large overhead and can lead to slowdown. This also requires that the interconnect supports remote interrupts.

- CPU0 can detect its lack of forward progress and flush its WPC entries. This solution is good since it does not introduce any processor-to-processor interrupt related overhead. However, it is hard to detect lack of forward progress. CPU0 spins on a flag not visible to the runtime system, that is, the protocol code never executes. A counter register or similar hardware feature might be used. The protocol code periodically updates the counter. Lack of forward progress is detected if the counter reaches zero.

A runtime system as described in Section 5.2 allows us to correctly and efficiently run all the applications in the SPLASH-2 benchmarks suite. However, we have manually inserted WPC flush code into `barnes` and `fmm` because they use user level synchronization not visible to the runtime system.

7. RELATED WORK

The Check-In/Check-Out (CICO) cooperative shared memory implementation presented by Hill et. al. [30] uses similar ideas as the WPC technique. CICO is a programming model where a programmer can reason about access time to memory and give simple hardware coherence protocol performance hints. A *check-out* annotation marks the expected first use and a *check-in* annotation terminates the expected use of the data. Whereas CICO annotations are inserted as hints, a WPC entry actually "checks-out" write permission since the directory/MTAG lock is not released until the next synchronization point or a WPC miss.

Shasta [11] uses *batching* of miss checks, that is a "static merge" of coherence actions at instrumentation time. For a sequence of shared loads and stores, that touches the same coherence unit, the Shasta system combines/eliminates some of the access control checks (if possible). This way, all of the loads and stores in this sequence can proceed with only one check. The current WPC implementation works as a dynamic version of Shasta's batching technique.

It would be interesting to investigate how Shasta batching and a WPC implementation can be combined. Consider a system that batches load and store in-line checks at instrumentation time. The WPC mechanism can be useful where the static analysis fails to batch together multiple coherence operations. In particular, in tight loops or other places where one can expect high WPC hit rate.

Shasta is a traditional fine-grain SW-DSM system that uses poll-based asynchronous interrupts. This implies that Shasta does not have to lock directory nor MTAG entries to check for permission. Instead, they use a private state table where a single bit indicates if a processor has write permission to a particular coherence unit. That is, they use a load instead of a lock before each store to shared memory. We have run sequential tests that show that an `ldub` instead of an `ldstub`[2] significantly decreases the store instrumentation overhead. However, since a WPC code snippet does not use any extra memory accesses it is able to outperform even a private state table based store snippet. Our results indicate that WPC outperforms load-based tests in some applications whereas it is outperformed in other. In other words, also a system with a non-blocking coherence protocol may gain performance by a WPC when spatial store locality is high.

8. CONCLUSIONS

In this paper, we introduce and evaluate a new *write permission cache* (WPC) technique that exploits spatial store locality. We demonstrate that the instrumentation overhead of the fine-grained software DSM (SW-DSM) system, DSZOOM [12], can be reduced with both 1- and 2-entry WPC implementations. On average, the original store instrumentation overhead, the single largest source of the total instrumentation cost, is reduced with 42 (67) percent for highly optimized (non-optimized) code. The parallel performance of the DSZOOM system for 16-processor runs (2-node configuration) of SPLASH-2 benchmarks is reduced by 7 (11) percent. We believe that instrumentation-time batching (Shasta's approach [11]) of coherence actions combined with our new WPC technique might improve performance even further.

9. REFERENCES

[1] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. thesis, Department of Computer Science, Yale University, Sept. 1986.

[2] L. Iftode and J. P. Singh, "Shared Virtual Memory: Progress and Challenges," *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, vol. 87, no. 3, pp. 498–507, Mar. 1999.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, Oct. 1991, pp. 152–164.

[2] `ldub` is load unsigned byte whereas `ldstub` is an atomic load store unsigned byte.

[4] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *ISCA92*, May 1992, pp. 13–21.

[5] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory," in *Proceedings of Operating Systems Design and Implementation Symposium*, Oct. 1996.

[6] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network," in *Proceedings of the 16th ACM Symposium on Operating System Principle*, Oct. 1997.

[7] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the Winter 1994 USENIX Conference*, Jan. 1994, pp. 115–131.

[8] H. Lu, A. L. Cox, and W. Zwaenepoel, "Contention Elimination by Replication of Sequential Sections in Distributed Shared Memory Programs," in *Proceedings of the eight conference on Principles and Practice of Parallel Programming*, June 2001.

[9] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh, "Home-based SVM protocols for SMP clusters: Design and Performance," in *Proceedings of the 4th International Symposium on High-Performance Computer Architeture (HPCA-4)*, Feb. 1998.

[10] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain Access Control for Distributed Shared Memory," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, Oct. 1994, pp. 297–306.

[11] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Oct. 1996, pp. 174–185.

[12] Z. Radović and E. Hagersten, "Removing the Overhead from Software-Based Shared Memory," in *Proceedings of Supercomputing 2001*, Denver, Colorado, USA, Nov. 2001.

[13] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sept. 1979.

[14] O. Grenholm, Z. Radović, and E. Hagersten, "Latency-hiding and Optimizations of the DSZOOM Instrumentation System," Tech. Rep. 2003-029, Department of Information Technology, Uppsala University, May 2003.

[15] J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," in *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 291–300.

[16] D. J. Scales, K. Gharachorloo, and A. Aggarwal, "Fine-Grain Software Distributed Shared Memory on SMP Clusters," Tech. Rep. 97/3, Western Research Laboratory, Digital Equipment Corporation, Feb. 1997.

[17] D. J. Scales and K. Gharachorloo, "Design and Performance of the Shasta Distributed Shared Memory Protocol," in *Proceedings of the 11th ACM International Conference on Supercomputing*, July 1997, Extended version available as Technical Report 97/2, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[18] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood, "Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations," Tech. Rep. 1307, Computer Sciences Department, University of Wisconsin–Madison, Mar. 1996.

[19] R. B. Gillett, "MEMORY CHANNEL Network for PCI," *IEEE Micro*, vol. 16, Feb. 1996.

[20] M. Fillo and R. B. Gillett, "Architecture and Implementation of MEMORY CHANNEL 2," *DEC Technical Journal*, vol. 9, pp. 27–41, July 1997.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, pp. 29–36, Feb. 1995.

[22] "InfiniBand(SM) Trade Association, InfiniBand Architecture Specification, Release 1.0," Oct. 2000, Available from: http://www.infinibandta.org.

[23] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," in *Proceedings of the 5th International Symposium on High-Performance Computer Architeture (HPCA-5)*, Feb. 1999, pp. 172–181.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, June 1995, pp. 24–36.

[25] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood, "Fast and Portable Parallel Architecture Simulators: Wisconsin Wind Tunnel II," *IEEE Concurrency*, 2000.

[26] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 1988, pp. 4–11, ACM Press.

[27] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres, "Gigaplane: A High Performance Bus for Large SMPs," in *Proceedings of IEEE Hot Interconnects IV*, Aug. 1996, pp. 41–52.

[28] L. W. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the 1996 USENIX Annual Technical Conference*, Jan. 1996, pp. 279–294.

[29] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3rd edition, 2003.

[30] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*. 1992, pp. 262–273, ACM Press.

A. WPC HIT RATE DATA

This Section shows hit rate for 1, 2, 3, 4, 8, 16, 32 and 1024 WPC entries when coherence unit size is varied from 32 to 8192 bytes. Figures 10 to 22 contain WPC data for the studied applications when run on 16 processors in protocol analyzer. The data is collected during the parallel execution phase.

B. PROCESSOR COUNT IMPACT ON WPC HIT RATE

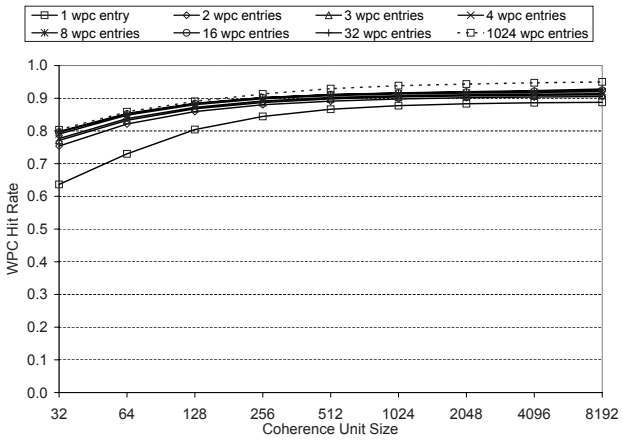
We have scaled the number of processors from 1 to 16 to investigate if the different data sets that the processors has to handle introduce any differences in WPC hit rate. The results indicate that the WPC hit rate is almost identical when the processor count is varied.

C. PARALLEL EXECUTION TIME

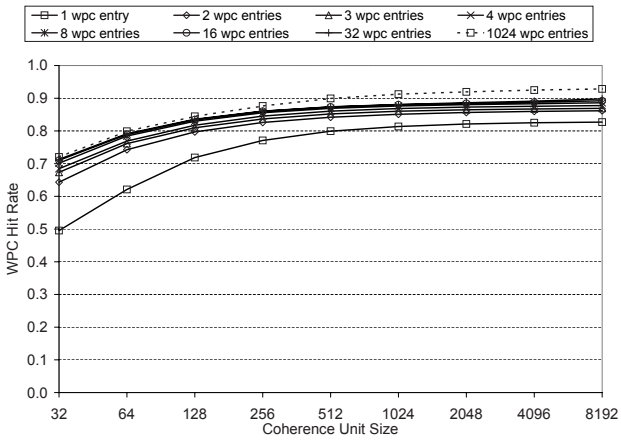
Figure 23 shows execution time for Sun Enterprise E6000 (SMP), 2-node Sun WildFire (HW-DSM) and three DSZOOM configurations:

1. DSZOOM-base: the original DSZOOM implementation.
2. DSZOOM-swpc: the DSZOOM implementation with a 1-entry WPC.
3. DSZOOM-dwpc: the DSZOOM implementation with a 2-entry WPC.

All DSZOOM configurations use a coherence unit size of 64 bytes. Both the HW-DSM configuration and the DSZOOM configurations run on two nodes and with eight processors per node (16 in total).

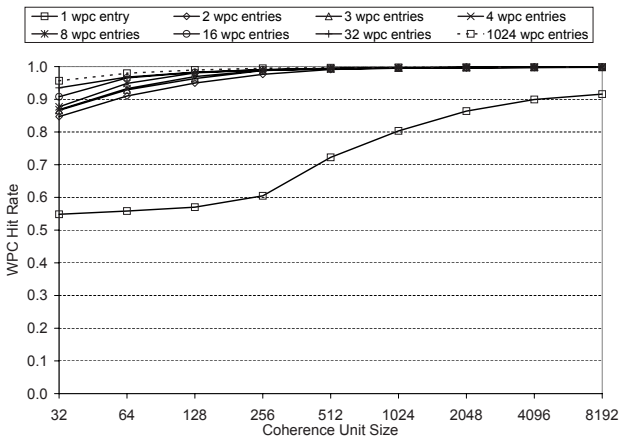


(a) Optimization level `-O0`.

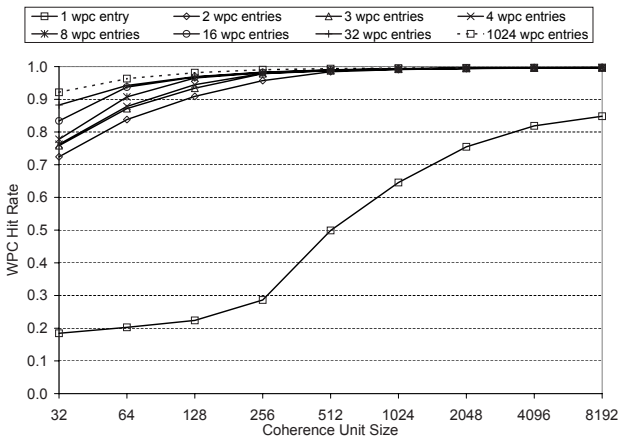


(b) Optimization level `-O3`.

Fig. 10. WPC hit rate for `barnes`.

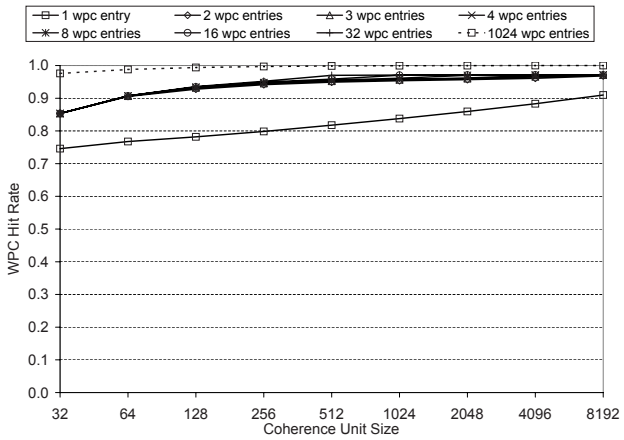


(a) Optimization level `-O0`.

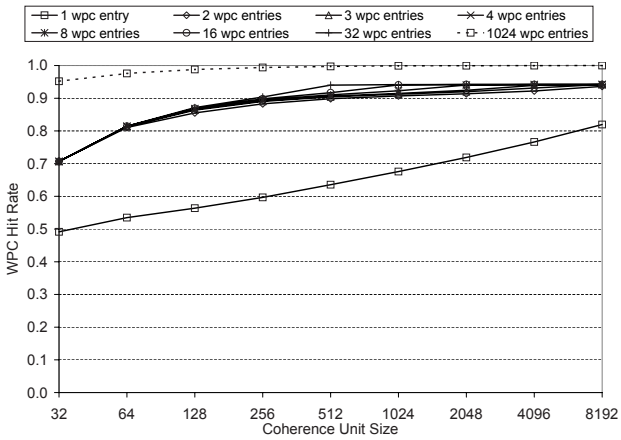


(b) Optimization level `-O3`.

Fig. 11. WPC hit rate for `cholesky`.

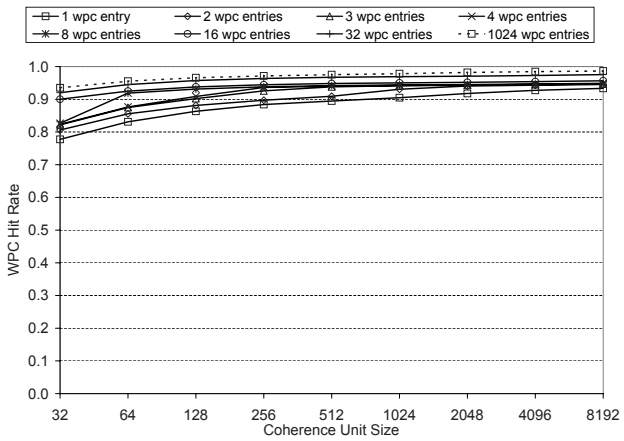


(a) Optimization level `-O0`.

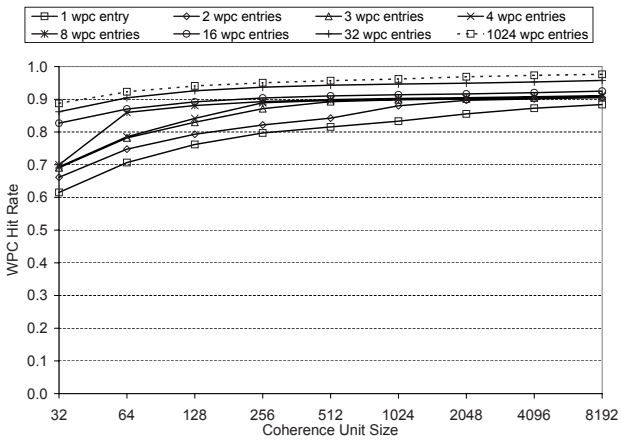


(b) Optimization level `-O3`.

Fig. 12. WPC hit rate for `fft`.

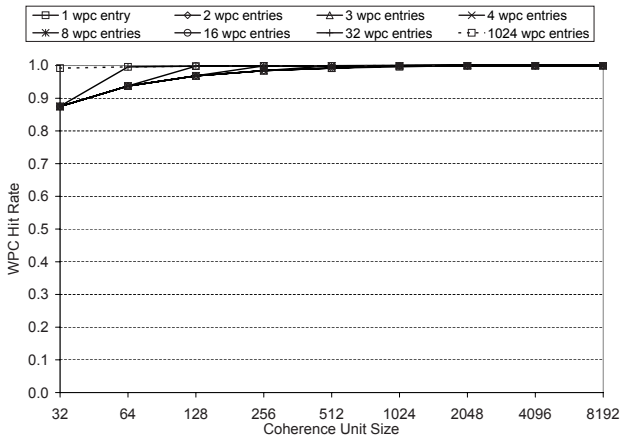


(a) Optimization level `-O0`.

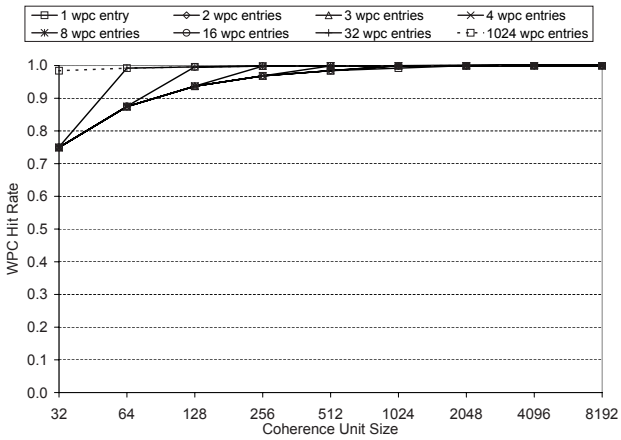


(b) Optimization level `-O3`.

Fig. 13. WPC hit rate for `fmm`.

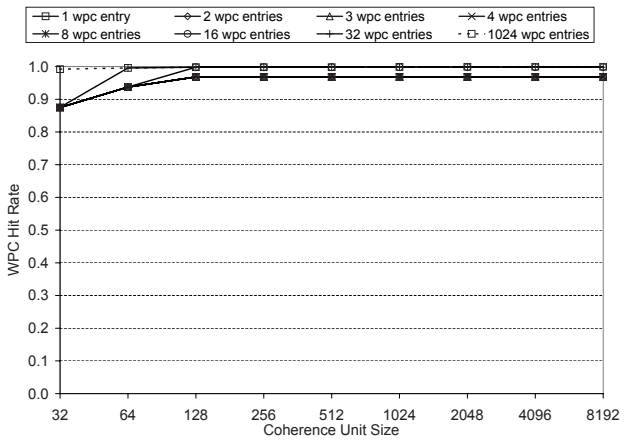


(a) Optimization level `-O0`.

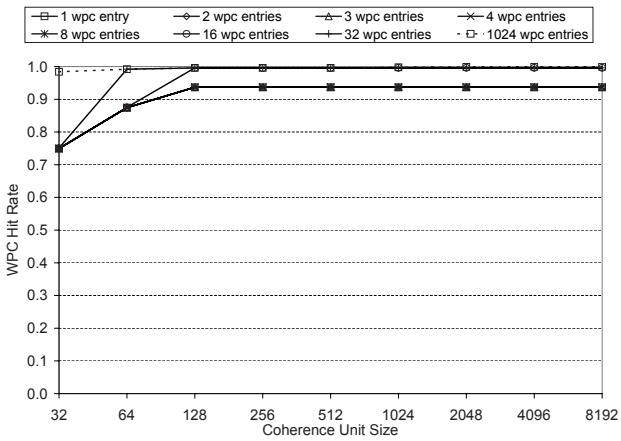


(b) Optimization level `-O3`.

Fig. 14. WPC hit rate for `lu-c`.

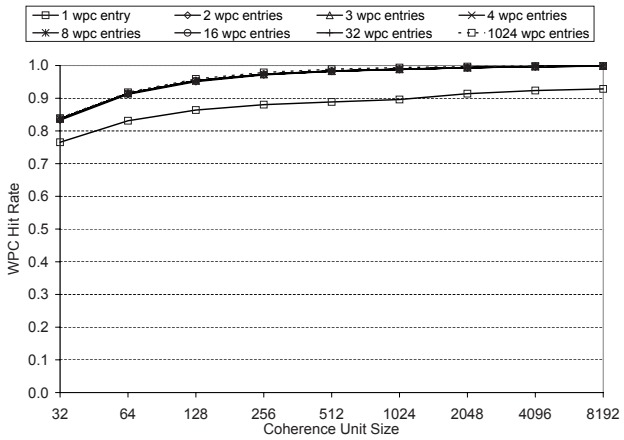


(a) Optimization level `-O0`.

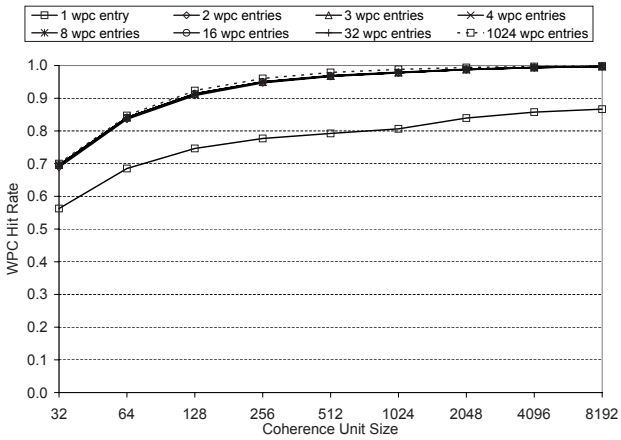


(b) Optimization level `-O3`.

Fig. 15. WPC hit rate for `lu-nc`.

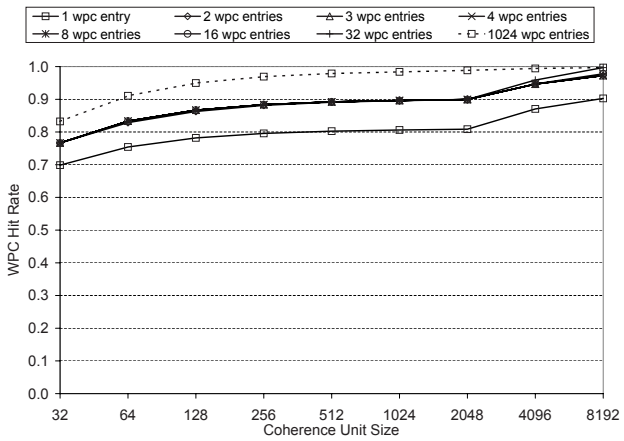


(a) Optimization level `-O0`.

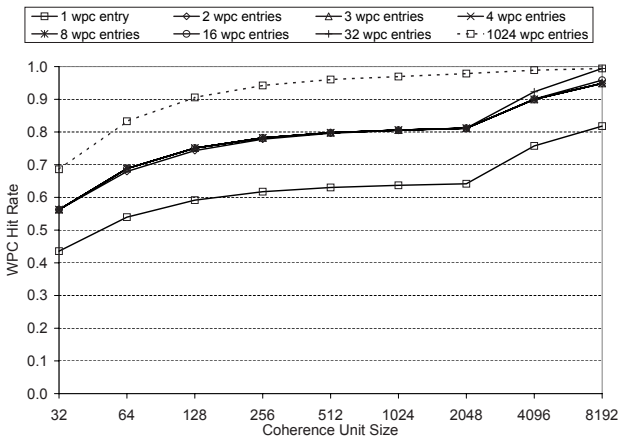


(b) Optimization level `-O3`.

Fig. 16. WPC hit rate for `ocean-c`.

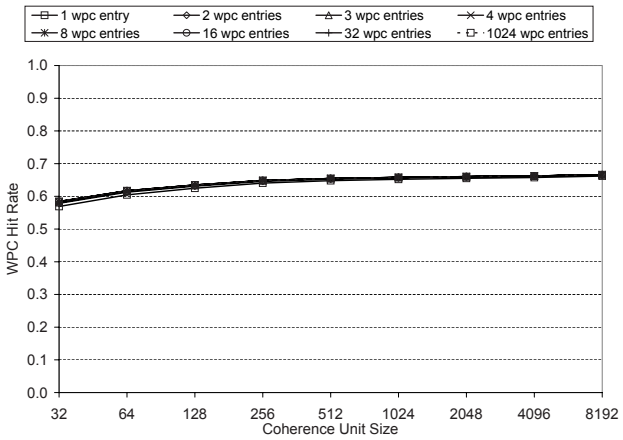


(a) Optimization level `-O0`.

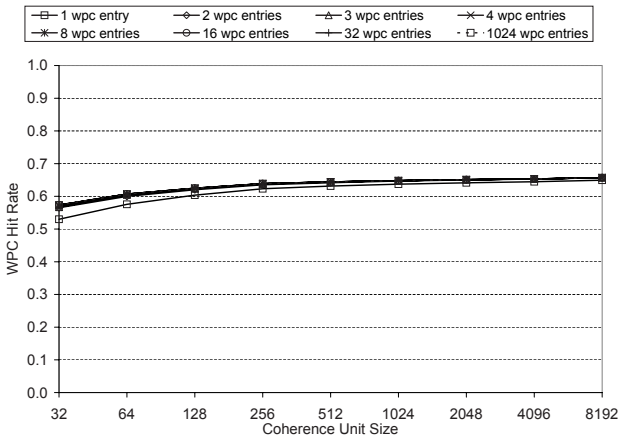


(b) Optimization level `-O3`.

Fig. 17. WPC hit rate for `ocean-nc`.

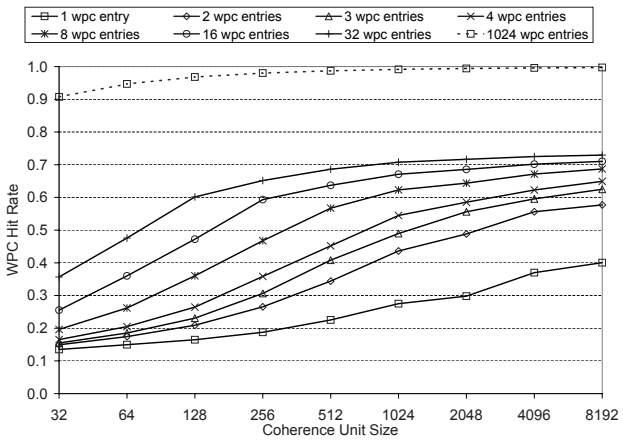


(a) Optimization level `-O0`.

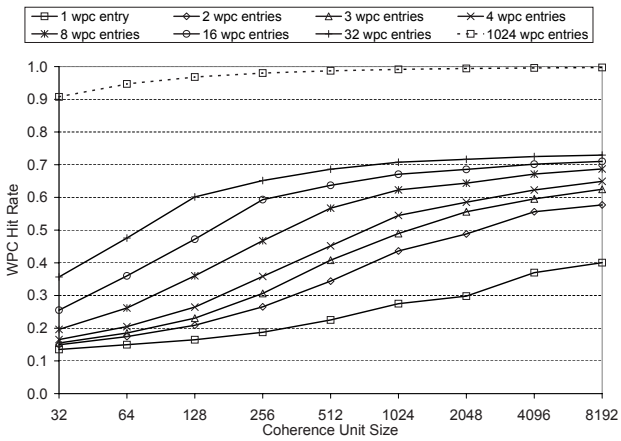


(b) Optimization level `-O3`.

Fig. 18. WPC hit rate for `radiosity`.

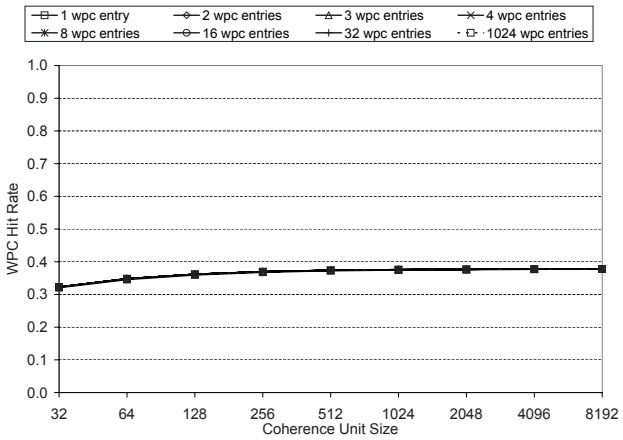


(a) Optimization level `-O0`.

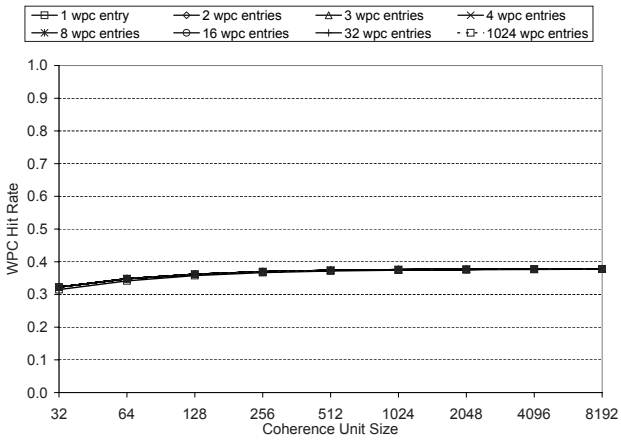


(b) Optimization level `-O3`.

Fig. 19. WPC hit rate for `radix`.

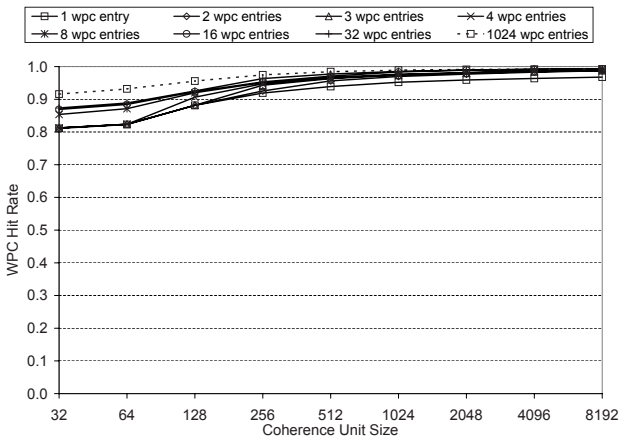


(a) Optimization level `-O0`.

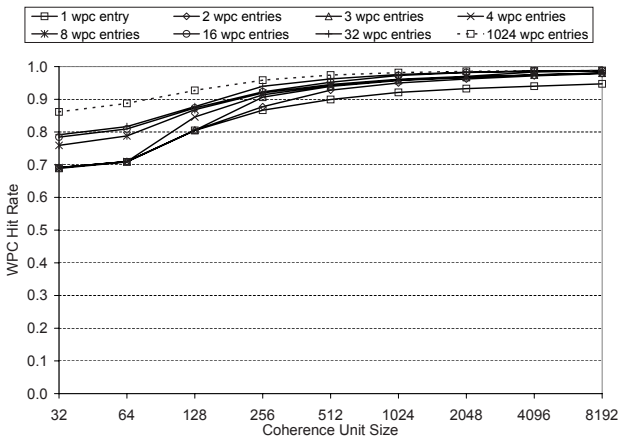


(b) Optimization level `-O3`.

Fig. 20. WPC hit rate for `raytrace`.

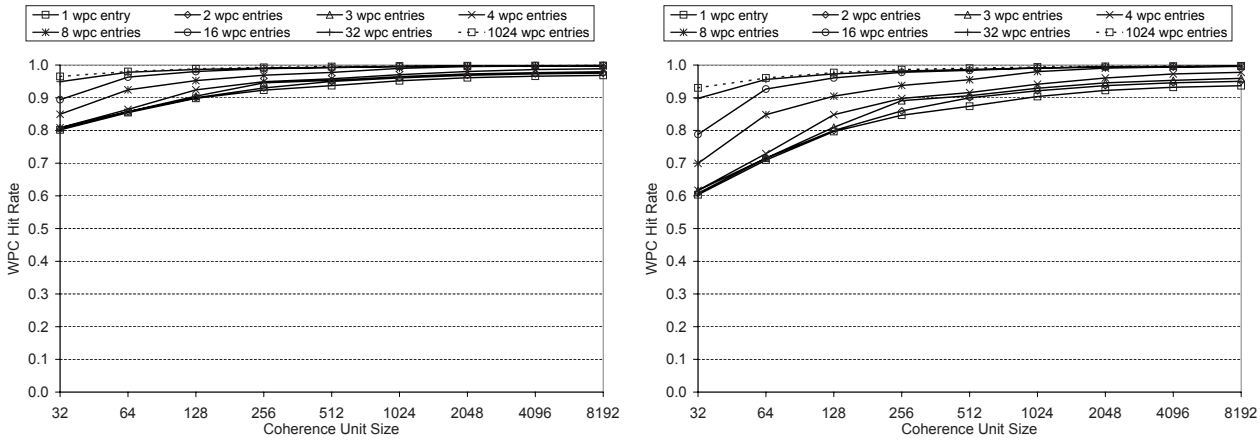


(a) Optimization level `-O0`.



(b) Optimization level `-O3`.

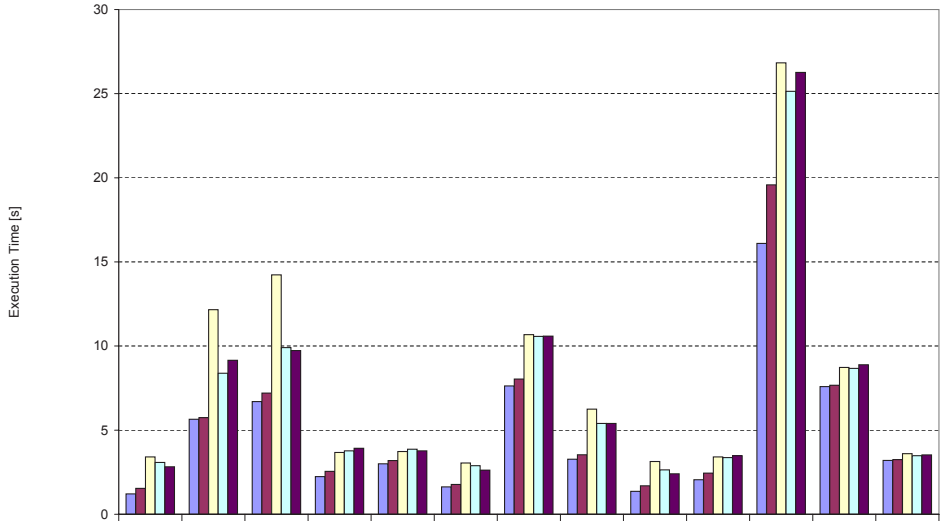
Fig. 21. WPC hit rate for `water-nsq`.



(a) Optimization level `-O0`.

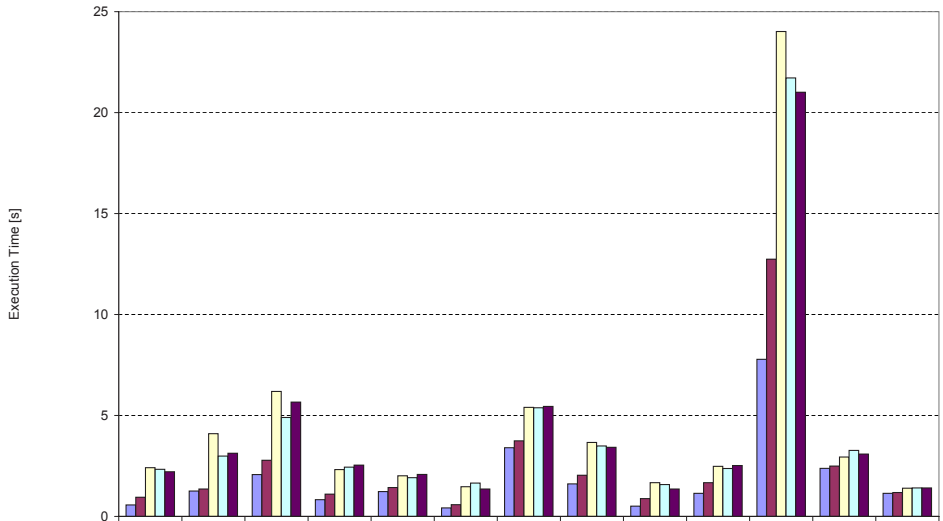
(b) Optimization level `-O3`.

Fig. 22. WPC hit rate for `water-sp`.



	fft	lu-c	lu-nc	radix	barnes	cholesky	fmm	ocean-c	ocean-nc	radiosity	raytrace	water-nsq	water-sp
SMP	1.21	5.65	6.70	2.24	3.00	1.63	7.63	3.27	1.36	2.05	16.10	7.59	3.20
HW-DSM	1.54	5.74	7.20	2.55	3.19	1.77	8.04	3.54	1.69	2.45	19.58	7.67	3.25
DSZOOM-base	3.41	12.16	14.23	3.67	3.73	3.05	10.68	6.25	3.14	3.41	26.83	8.73	3.60
DSZOOM-swpc	3.08	8.38	9.91	3.77	3.87	2.89	10.57	5.40	2.64	3.37	25.14	8.67	3.48
DSZOOM-dwpc	2.82	9.15	9.73	3.92	3.77	2.62	10.59	5.40	2.41	3.48	26.26	8.89	3.53

(a) 16-processor runs and non-optimized binaries.



	fft	lu-c	lu-nc	radix	barnes	cholesky	fmm	ocean-c	ocean-nc	radiosity	raytrace	water-nsq	water-sp
SMP	0.57	1.26	2.07	0.83	1.23	0.42	3.40	1.61	0.51	1.14	7.78	2.38	1.14
HW-DSM	0.95	1.36	2.78	1.10	1.43	0.58	3.74	2.04	0.88	1.67	12.74	2.49	1.18
DSZOOM-base	2.41	4.10	6.19	2.32	2.01	1.47	5.40	3.66	1.67	2.48	24.02	2.94	1.40
DSZOOM-swpc	2.33	2.99	4.90	2.44	1.92	1.65	5.38	3.49	1.58	2.37	21.72	3.27	1.41
DSZOOM-dwpc	2.21	3.13	5.66	2.54	2.08	1.36	5.45	3.42	1.36	2.52	21.01	3.09	1.41

(b) 16-processor runs and fully optimized binaries.

Fig. 23. Parallel execution time for 16-processor configurations.

Recent technical reports from the Department of Information Technology

2003-051 Pavel Krcal and Wang Yi: *Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata*

2003-052 Magnus Svärd and Jan Nordström: *Well Posed Boundary Conditions for the Navier-Stokes Equations*

2003-053 Erik Bängtsson and Maya Neytcheva: *Approaches to Reduce the Computational Cost when Solving Linear Systems of Equations Arising in Boundary Element Method Discretizations*

2003-054 Martin Nilsson: *Stability of the Fast Multipole Method for Helmholtz Equation in Three Dimensions*

2003-055 Martin Nilsson: *Rapid Solution of Parameter-Dependent Linear Systems for Electromagnetic Problems in the Frequency Domain*

2003-056 Parosh Aziz Abdulla, Johann Deneux, Pritha Mahata, and Aletta Nylén: *Forward Reachability Analysis of Timed Petri Nets*

2003-057 Erik Berg: *Low-Overhead Spatial and Temporal Data Locality Analysis*

2003-058 Erik Berg: *StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis*

2003-059 Jonas Persson and Lina von Sydow: *Pricing European Multi-asset Options Using a Space-time Adaptive FD-method*

2003-060 Pierre Flener: *Realism in Project-Based Software Engineering Courses: Rewards, Risks, and Recommendations*

2003-061 Lars Ferm and Per Lötstedt: *Space-Time Adaptive Solution of First Order PDEs*

2003-062 Emilio Tuosto, Björn Victor, and Kidane Yemane: *Polyadic History-Dependent Automata for the Fusion Calculus*

2003-063 Michael Baldamus, Joachim Parrow, and Björn Victor: *Spi Calculus Translated to π-Calculus Preserving May-Testing*

2003-064 Arnim Brüger, Bertil Gustafsson, Per Lötstedt, and Jonas Nilsson: *High Order Accurate Solution of the Incompressible Navier-Stokes Equations*

2003-065 Michael Baldamus, Richard Mayr, and Gerardo Schneider: *A Backward/Forward Strategy for Verifying Safety Properties of Infinite-State Systems*

2004-001 Torsten Söderström, Torbjörn Wigren, and Emad Abd-Elrady: *Maximum Likelihood Modeling of Orbits of Nonlinear ODEs*

2004-002 Pablo Giambiagi, Gerardo Schneider, and Frank D. Valencia: *On the Expressiveness of CCS-like Calculi*

2004-003 Johan Elf, Per Lötstedt, and Paul Sjöberg: *Problems of High Dimension in Molecular Biology*

2004-004 Torbjörn Wigren: *Recursive Prediction Error Identification of Nonlinear State Space Models*

2004-005 Håkan Zeffer, Zoran Radovic, Oskar Grenholm, and Erik Hagersten: *Evaluation, Implementation and Performance of Write Permission Caching in the DSZOOM System*

UPPSALA
UNIVERSITET

February (updated June) 2004
ISSN 1404-3203
http://www.it.uu.se/