FEBRUARY 1997

WRL Research Report 97/2

Performance of the Shasta Distributed Shared Memory Protocol

Daniel J. Scales
Kourosh Gharachorloo

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Performance of the Shasta Distributed Shared Memory Protocol

Daniel J. Scales
Kourosh Gharachorloo

February 1997

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Abstract

Shasta supports a shared address space in software across a cluster of computers with physically distributed memory. A unique aspect of Shasta compared to most other software distributed shared memory systems is that shared data can be kept coherent at a fine granularity. Shasta implements this coherence by inserting inline code that checks the cache state of shared data before each load or store. In addition, Shasta allows the coherence granularity to be varied across different shared data structures in a single application. This approach alleviates potential inefficiences that arise from the fixed large (page-size) granularity of communication typical in most software shared memory systems.

This paper focuses on the design and performance of the cache coherence protocol in Shasta. Since Shasta is implemented entirely in software, it provides tremendous flexibility in the design of the cache coherence protocol. We have implemented an efficient cache coherence protocol that incorporates a number of optimizations, some of which are common in hardware shared memory systems. The above protocol is fully functional and runs on a prototype cluster of Alpha systems connected through Digital's Memory Channel network. To characterize the benefits of the various optimizations, we also present detailed performance results for nine SPLASH-2 applications running on this cluster.

1 Introduction

There has been much interest in distributed shared memory (DSM) systems that support a shared address space in software across a cluster of workstations. The most common approach, called Shared Virtual Memory (SVM), uses the virtual memory hardware to detect access to data that is not available locally [4, 14, 16]. These systems communicate data and maintain coherence at a fixed granularity equal to the size of a virtual page. As an alternative, a few software systems have explored the feasibility of maintaining coherence at a finer granularity [19, 21]. Support for fine-grain sharing is important for reducing false sharing and the transmission of unneeded data, both of which are potential problems in systems with large coherence granularities. Fine-grain access to shared data is supported by inserting code in an application executable before loads and stores that checks if the data being accessed is available locally in the appropriate state. Recent work in the context of the Shasta system has shown that the cost of the inline checks can be minimized by applying appropriate optimizations [19], making this approach a viable alternative to SVM systems.

The goal of this paper is to describe the design and performance of the cache coherence protocol in Shasta. Since Shasta supports the shared address space entirely in software, it provides a flexible framework for experimenting with a variety of cache coherence protocol optimizations to improve parallel performance. By supporting coherence at a fine granularity, Shasta alleviates the need for complex mechanisms for dealing with false sharing that are typically present in software page-based systems. Therefore, the basic cache coherence protocol in Shasta more closely resembles that of a hardware distributed shared memory system.

The Shasta coherence protocol provides a number of mechanisms for dealing with the long communication latencies in a workstation cluster. One of the unique aspects of the protocol (relative to other protocols that transparently support a shared address space) is its ability to support variable coherence granularities across different shared data structures within the same application. This feature enables Shasta to exploit any potential gains from larger communication granularities for specific shared data.

Given the relatively high overheads associated with handling messages in software, we have also designed the protocol to minimize extraneous coherence messages. Thus, our protocol typically requires fewer messages to satisfy shared memory operations compared to protocols commonly used in hardware DSM systems (e.g., DASH [15]). The protocol also includes optimizations, such as non-blocking stores, that aggressively exploit a relaxed memory consistency model. Other optimizations include detection of migratory data sharing, issuing multiple load misses simultaneously, merging of load and store misses to the same cache line, and support for prefetching and home placement directives.

The Shasta protocol has been implemented on our prototype cluster and is fully functional. The cluster consists of a total of sixteen 300 MHz Alpha processors connected through the Memory Channel [10]. We present detailed performance results for nine SPLASH-2 applications running on the above cluster. The results characterize the various overheads in parallel runs, including stalls for data, synchronization time, and time spent handling requests from other processors. In addition, we analyze the effects of the various protocol optimizations. Support for variable granularity is by far the most important optimization in Shasta, leading to performance improvements ranging from 10% to 95%. Surprisingly, optimizations that attempt to hide memory latency, such as exploiting a relaxed memory consistency model, lead to much more limited gains. A significant portion of the time while a processor is waiting for data or synchronization is overlapped with the handling of incoming coherence messages from other processors, thus making it difficult to improve performance by reducing the wait times. Finally, optimizations related to migratory data are not useful in Shasta primarily because migratory sharing patterns are unstable or not present at block sizes of 64 bytes or higher.

The following section describes the basic design of Shasta, including the inline state checks and the protocol that is invoked in case of a miss. Section 3 discusses optimizations to the basic cache coherence protocol. We present detailed performance results in Section 4. Finally, we describe related work and conclude.

2 Basic Design of Shasta

In this section, we present an overview of the base Shasta system, which is described more fully in a previous paper [19]. Shasta divides the virtual address space of each processor into private and shared regions. Data in the shared region may be cached by multiple processors at the same time, with copies residing at the same virtual address on each processor. The base Shasta system adopts the memory model of the original SPLASH applications [22]: data that is dynamically allocated is shared, but all static and stack data is private.

2.1 Cache Coherence Protocol

As in hardware cache-coherent multiprocessors, shared data in the Shasta system has three basic states:

- invalid - the data is not valid on this processor.
- shared - the data is valid on this processor, and other processors have copies of the data as well.
- exclusive - the data is valid on this processor, and no other processors have copies of this data.

Communication is required if a processor attempts to read data that is in the invalid state, or attempts to write data that is in the invalid or shared state. In this case, we say that there is a *shared miss*. The checks that Shasta inserts in the application executables at each load and store are shared miss checks on the data being referenced.

As in hardware shared-memory systems, Shasta divides up the shared address space into ranges of memory, called *blocks*. All data within a block is in the same state and is always fetched and kept coherent as a unit. A unique aspect of the Shasta system is that the block size can be different for different ranges of the shared address space (i.e., for different program data). To simplify the inline code, Shasta divides up the address space further into fixed-size ranges called *lines* and maintains state information for each line in a *state table*. The line size is configurable at compile time and is typically set to 64 or 128 bytes. The size of each block must be a multiple of the fixed line size.

Coherence is maintained using a directory-based invalidation protocol. The protocol supports three types of requests: *read*, *read-exclusive*, and *exclusive* (or *upgrade*). Supporting exclusive requests is an important optimization since it reduces message latency and overhead if the requesting processor already has the line in shared state. Shasta also currently supports three types of synchronization primitives in the protocol: locks, barriers, and event flags. These primitives are sufficient for supporting the SPLASH-2 applications.

A home processor is associated with each virtual page of shared data, and each processor maintains *directory* information for the shared data pages assigned to it. The protocol maintains the notion of an *owner* processor for each line, which corresponds to the last processor that maintained an exclusive copy of the line. The directory information consists of two components: (i) a pointer to the current owner processor, and (ii) a full bit vector of the processors that are sharing the data. Our protocol supports *dirty sharing*, which allows the data to be shared without requiring the home node to have an up-to-date copy. A request that arrives at the home is always forwarded to the current owner; as an optimization, this forwarding is avoided if the home processor has a copy of the data.

Because of the high cost of handling messages via interrupts, messages from other processors are serviced through a polling mechanism. The base Shasta implementation polls for incoming messages whenever the protocol waits for a reply. To ensure reasonable response times, Shasta also insert polls at every loop backedge. Polling is inexpensive (three instructions) in our Memory Channel cluster because the implementation arranges for a single cachable location that can be tested to determine if a message has arrived. The use of polling also simplifies the inline miss checks, since the Shasta compiler ensures that there is no handling of messages between a shared miss check and the load or store that is being checked.

2.2 Basic Shared Miss Check

Figure 1 shows Alpha assembly code that does a store miss check. This code first checks if the target address is in the shared memory range and if not, skips the remainder of the check. Otherwise, the code calculates the address of the state table entry corresponding to the target address and checks that the line containing the target address is in the exclusive state. This code has been optimized in a number of ways. For example, the code does not save or restore registers. The Shasta compiler does live register analysis to find unused registers (labeled `rx` and `ry` in the figure) at the point where it inserts the miss check. Shasta does not need to check accesses to non-shared (i.e., private) data,

```
      lda      rx, offset(base)
      srl      rx, SHARED_HEAP_BITS, ry
      srl      rx, LINE_BITS, rx
      beq      ry, nomiss
      ldq_u    ry, 0(rx)
      extbl    ry, rx, ry
      beq      ry, nomiss

      ...call function to handle store miss

   nomiss:
      ... store instruction
```

Figure 1: Store miss check code.

which includes all stack and static data in the current implementation. Therefore, a load or store whose base register uses the stack pointer (SP) or global pointer (GP) register, or is calculated using the contents of the SP or GP, is not checked.

Despite the simple optimizations applied to the basic checks, the overhead of the miss checks can be significant for many applications, often approaching or exceeding 100% of the original sequential execution time. The Shasta system applies a number of more advanced optimizations that dramatically reduce this overhead to an average of about 20% (including polling overhead) across the SPLASH-2 applications [19].[1] The two most important optimizations are described below.

Invalid Flag Technique. Whenever a line on a processor becomes invalid, the Shasta protocol stores a particular "flag" value in each longword (4 bytes) of the line. The miss check code for a load can then just compare the value loaded with the flag value. If the loaded value is not equal to the flag value, the data must be valid and the application code can continue immediately. If the loaded value is equal to the flag value, then a miss routine is called that first does the normal range check and state table lookup. The state check distinguishes an actual miss from a "false miss" (i.e., when the application data actually contains the flag value), and simply returns back to the application code in case of a false miss. Since false misses almost never occur in practice, the above technique can greatly reduce the load miss check overhead. Another advantage of the invalid flag technique is that the load of the state table entry is eliminated. Therefore, there are no additional data cache misses beyond what would occur in the application code.

Batching Miss Checks. A very important technique for reducing the overhead of miss checks is to batch together checks for multiple loads and stores. Suppose there are a sequence of loads and stores that are all relative to the same (unmodified) base register and the offsets (with respect to the base register) span a range whose length is less than or equal to the Shasta line size. These loads and stores can collectively touch at most two consecutive lines in memory. Therefore, if inline checks verify that these are in the correct state, then all the loads and stores can proceed without further checks. One convenient way to check both lines is to do a normal shared miss check on the beginning address and ending address of the range. Fortunately, the checking code for the two endpoints can be interleaved effectively to eliminate pipeline stalls; therefore, the cycle count is less than double the cycle count of normal checks. The batching technique also applies to loads and stores via multiple base registers. For each set of loads and stores that can be batched, the Shasta compiler generates code to check the lines that may be referenced via each base register that is used. A batch miss handling routine is called if any of the lines referenced via any of the base registers are not in the correct state. Batching can also be useful for eliminating and hiding communication latency in a parallel execution, since it allows load and store misses to the same line to be combined into a single store miss and misses on multiple lines to be serviced at the same time.

[1] The relative effect of the checking overhead is typically less on the parallel execution time due to other overheads arising from communication and synchronization. For example, consider an application run that achieves a parallel efficiency of 50% without any checking overhead. Given a checking overhead of 20% on a uniprocessor run, the effective overhead on the parallel execution time is reduced to 10%.

3 Protocol Optimizations

This section describes a number of the optimizations in the Shasta coherence protocol that attempt to reduce the effect of the long latencies and large message overheads that are typical in software DSM systems. The optimizations include minimizing extraneous protocol messages, supporting prefetch and home placement directives, supporting coherence and communication at multiple granularities, exploiting a relaxed memory model, batching together requests for multiple misses, and optimizing accesses to migratory data.

3.1 Minimizing Protocol Messages

Given the relatively high overheads associated with handling messages in software DSM implementations, we have designed our protocol to minimize extraneous coherence messages.

A key property of our protocol is that the current owner node specified by the directory guarantees to service a request that is forwarded to it. We exploit the flexibility of our software protocol to guarantee this property. First, the main memory at each node acts as a software-controlled cache of remote data and there are no forced writebacks or replacements in our protocol, allowing the current owner to maintain a valid copy of the data. Second, there is no need to retry requests (e.g., by sending a negative-acknowledgement reply) due to either transient states or deadlock conditions. We can always allocate queue space at the target processor to delay servicing an incoming request in transient cases, and there are no deadlock conditions that necessitate a retry mechanism (such as limited network buffer space in hardware DSM systems).

The fact that the current owner guarantees to service a forwarded request allows the protocol to complete all directory state changes when a request first reaches the home. This property eliminates the need for extra messages that are sent back to the home to confirm that the forwarded request is satisfied (e.g., "ownership change" or "sharing writeback" messages that are common in hardware DSM protocols such as DASH [15]). Therefore, our protocol can handle three-hop transactions involving a remote owner more efficiently.

We use several other techniques to reduce the number or size of protocol messages. The fact that the protocol supports dirty sharing eliminates the need for sending an up-to-date copy of the line back to the home for three-hop read transactions (i.e., when the home node is remote and the data is dirty in yet another node).[2] Supporting exclusive (or upgrade) requests is also an important optimization since it reduces the need for fetching data on a store if the requesting processor already has the line in shared state. Finally, the number of invalidation acknowledgements that are expected for an exclusive request is piggybacked on one of the invalidation acknowledgements to the requestor instead of being sent as a separate message.

3.2 Multiple Coherence Granularity

The most novel aspect of our protocol is its ability to support multiple granularities for communication and coherence, even within a single application. The fact that the basic granularity for the inline state check is software configurable already gives us the ability to use different granularities for different applications. Nevertheless, the ability to further vary the communication and coherence granularity within a single application can provide a significant performance boost in a software DSM system, since data with good spatial locality can be communicated at a coarse grain to amortize large communication overheads, while data prone to false sharing can use a finer sharing granularity.

Our current implementation automatically chooses a block size based on the allocated size of a data structure. Our basic heuristic is to choose a block size equal to the object size up to a certain threshold; the block size for objects larger than a given threshold is simply set to the base Shasta line size (typically set to be 64 bytes). The rationale for the heuristic is that small objects should be transferred as a single unit, while larger objects (e.g., large arrays) should be communicated at a fine granularity to avoid false sharing. We also allow the programmer to override this heuristic by providing a special version of `malloc` that takes a block size as an argument (the block size must be a multiple of the base Shasta line size). Since the choice of the block size does not affect the correctness of the program, the programmer can freely experiment with various block sizes (for the key data structures) to tune the performance of an

[2] Our protocol supports sharing writeback messages as an option, however.

application. Controlling the coherence granularity in this manner is significantly simpler than approaches adopted by object- or region-based DSM systems [3, 12, 17, 20], since the latter approaches can affect correctness and typically require a more substantial change to the application.

We currently associate different granularities to different virtual pages and place newly allocated data on the appropriate page. The block size for each page is communicated to all the nodes at the time the pool of shared pages are allocated. To determine the block size of data at a particular address, a requesting processor simply checks the block size for the associated page. The above mechanism is simple yet effective for most applications. We are also working on a more general and dynamic mechanism that maintains the block size information permanently only at the home along with the directory information for each line [19].

Exploiting variable granularity may reduce the relative gains from other optimizations if a larger block size is effective at eliminating many of the misses on shared data structures.

3.3 Prefetch and Home Placement Directives

The Shasta protocol allows the application to explicitly specify the home processor for individual pages instead of relying on the default round-robin allocation. The protocol also supports non-binding prefetch and prefetch-exclusive directives. The Shasta system can optionally supply information on source code lines that suffer the most number of remote misses by keeping extra state within the protocol. The programmer can use this information to identify places where prefetching may be helpful.

3.4 Exploiting Relaxed Memory Models

Our protocol aggressively exploits the release consistency model [9] by emulating the behavior of a processor with non-blocking loads and stores and a lockup-free cache. Because of our non-blocking load and store operations, a line may be in one of two pending states, *pending-invalid* and *pending-shared*. The pending-invalid state corresponds to an outstanding read or read-exclusive request on that line; pending-shared signifies an outstanding exclusive request. The protocol supports *non-blocking stores* by simply issuing a read-exclusive or exclusive request, recording where the store occurred, and continuing. This information allows the protocol to appropriately merge the reply data with the newly written data that is already in memory. Our protocol also exhibits a limited form of *non-blocking load* behavior due to the batching optimization, since batching can lead to multiple outstanding loads (as described in Section 2.2). Finally, we support *non-blocking releases* by delaying a release operation (e.g., unlock) on the side until all previous operations are complete (analogous to placing the release in a write buffer in hardware implementations), and allowing the processor to continue with operations following the release.

We also support aggressive *lockup-free* behavior for lines that are in a pending state. Writes to a pending line are allowed to proceed by storing the newly written data into memory and recording the location of the stores in the miss handler (invoked due to the pending state). Loads from a line in pending-shared state are allowed to proceed immediately, since the node already has a copy of the data. Loads from a line in pending-invalid state are also allowed to proceed as long as the load is from a valid section of the line. The above two cases are well-suited to the “flag” check for loads since this technique can efficiently detect a “hit” in both cases without actually checking the state for the line. Finally, we support *eager exclusive replies* in the case of a read-exclusive request by sending the reply data back to the requesting processor as soon as possible and allowing it to use the data immediately (by setting the local state to exclusive), even though requests from other processors are delayed until all pending invalidations are acknowledged.

3.5 Batching

The batching technique described in Section 2.2 can reduce communication overhead by merging load and store misses to the same line and by issuing requests for multiple lines at the same time. Our protocol handles a miss associated with the batching of loads and stores as follows. The batch checking code jumps to the inline batch miss code if there is a miss on any line within a batch. The inline code calls a batch miss handler that issues all the necessary miss requests. We implement non-stalling stores by requiring the handler to wait only for outstanding read and read-exclusive replies and not for invalidation acknowledgements.

Although the batch miss handler brings in all the necessary lines, it cannot guarantee that all the lines will be in the appropriate state once all the replies have come back. The reason is that while the handler is waiting for the replies, requests from other processes must be served to avoid deadlock; these requests can in turn change the state of the lines within the batch. Even though a line may not be in the right state, loads to the line will still get the correct value (under release consistency) as long as the original contents of the line remain in memory. We therefore delay storing the flag value into memory for invalidated lines until after the end of the batch. After the batch code has been executed, we complete the invalidation of any such lines at the time of the next entry into the protocol code (due to polls, misses, or synchronization). We may also have to reissue stores to lines which are no longer in exclusive or pending-shared state before we start the batch code. A relaxed memory model simplifies handling the above corner cases in an efficient manner.

3.6 Detecting Migratory Sharing Patterns

The Shasta protocol provides a sophisticated mechanism for detecting data that is shared in a migratory fashion and optimizing accesses to such data [6, 23]. Migratory sharing occurs when data is read and modified by different processors, leading to the migration of the data from one processor to another. By keeping extra information at each directory entry, the protocol detects whether the data in each line exhibits migratory behavior. A line is designated for migratory conversion after the migratory sharing pattern is successfully observed for a threshold number of times. A read request to a line that is designated for migratory conversion is automatically converted to a read-exclusive request at the directory. This conversion avoids the load miss followed by a store miss to the same line that is typical for migratory shared data. The protocol provides a mechanism to *revert* a line from migratory conversion. The reply data for a converted read request is cached with a special caching state (called exclusive-migratory). Operations by the owner processor treat the line as exclusive, and a subsequent store by that processor changes the line to the ordinary exclusive state. The protocol detects a break in the migratory behavior if an incoming request from another processor arrives before the owner processor writes to the line (i.e., while line is still in exclusive-migratory state). In this case, a message is sent to the home directory to nullify or revert the migratory conversion for that line. The line may subsequently be designated for migratory conversion if migratory behavior is observed again. The protocol provides hysteresis to avoid continuous switching to and from the migratory conversion state, and can optionally stop switching to migratory conversion for a given line if the line reverts a threshold number of times.

4 Performance Results

This section presents performance results for the Shasta implementation. We first describe our prototype cluster and the applications used in our study. We then present detailed performance results that show the performance of Shasta and the effectiveness of the various protocol optimizations described in the previous section.

4.1 Prototype Cluster

Our cluster consists of four AlphaServer 4100s connected by a Memory Channel network. Each AlphaServer 4100 has four 300 MHz 21164 processors, which each have 16 Kbyte on-chip instruction and data caches, a 96 Kbyte on-chip combined second-level cache, and a 2 Mbyte board-level cache. The individual processors are rated at 8.11 SpecInt95 and 12.7 SpecFP95, and the system bus has a bandwidth of 1 Gbyte/s. The Memory Channel is a memory-mapped network that allows a process to transmit data to a remote process without any operating system overhead via a simple store to a mapped page [10]. The one-way latency from user process to user process over Memory Channel is about 4 microseconds, and each network link can support a bandwidth of 60 MBytes/sec.

We have implemented a message-passing layer that runs efficiently on top of the Memory Channel. By using separate message buffers between each pair of processors, we avoid the need for any locking when adding or removing messages from the buffers. In Shasta, the minimum latency to fetch a 64-byte block from a remote processor (two hops) via the Memory Channel is 20 microseconds, and the effective bandwidth for large blocks is about 35 Mbytes/s.

	problem size	sequential time	with Shasta miss checks	miss check overhead
Barnes	16K particles	9.05s	9.92s	9.6%
FMM	32K particles	13.55s	15.36s	13.4%
LU	1024x1024 matrix	27.32s	29.59s	8.3%
LU-Contig	1024x1024 matrix	17.51s	21.99s	25.6%
Ocean	514x514 ocean	11.04s	13.29s	20.5%
Raytrace	balls4	71.53s	79.59s	11.3%
Volrend	head	1.62s	1.76s	8.6%
Water-Nsq	1000 molecules	7.87s	9.21s	17.0%
Water-Sp	1728 molecules	6.68s	7.64s	14.4%

Table 1: Sequential times and checking overheads for the SPLASH-2 applications.

We exploit message passing through shared memory segments when the communicating processors are on the same node; the minimum latency for a 64-byte block fetch within a node is 11 microseconds, and the bandwidth is about 45 Mbytes/s. Parallel runs of up to 4 processors use a single processor per node; the 8 and 16 processor runs use 2 and 4 processors per node, respectively. [3]

4.2 Applications

We report results for nine of the SPLASH-2 applications [24]. Table 1 shows the input sizes used in our experiments along with the sequential running times. We have increased the standard input sizes in order to make sure that the applications run for at least a few seconds on our cluster. Table 1 also shows the single processor execution times for each application after the Shasta miss checks are added (includes poll instructions), along with the percentage increase in the time over the original sequential time. The checking overheads range from 10% to 25%; the relative effect of this overhead is typically less on parallel execution times due to other overheads arising from communication and synchronization.

4.3 Parallel Performance

This section presents the parallel performance of the applications in Shasta, and analyzes the effectiveness of supporting variable coherence granularities within the protocol.

The following specifies the parameters used for the base set of experiments. We use a fixed Shasta line size of 64 bytes. Unless specified otherwise, the block size of objects less than 1024 bytes is automatically set to the size of the object, while larger objects use a 64-byte block size. Except for Water-Sp which ends up mainly using 1024-byte blocks, the remaining applications are virtually unaffected by this heuristic and use 64-byte blocks. In addition, for LU-Contiguous and Ocean, we use the home placement optimization as is done in most studies of the SPLASH-2 applications. We use all the optimizations related to exploiting release consistency (described in Section 3.4) except for non-blocking releases. We do not use the sharing writeback option (i.e., home is not updated on a 3-hop read operation). Finally, the migratory optimizations are turned off and we do not use any prefetch directives.[4]

Figure 2 shows the speedups for the applications running on our prototype cluster. The speedups shown are based on the execution time of the application running via Shasta on 1 to 16 processors relative to the execution of the original sequential application (with no miss checks). The speedup curves on the left side of the figure represent runs with the default 64-byte block size, except for Water-Sp which automatically uses 1024-byte blocks for most data structures.

[3] Results for 2, 4 and 8 processors are not directly comparable with those presented in in another Shasta paper [18] because of different assignment of processors to nodes.

[4] We do not yet have a comprehensive set of results with prefetching. Automatic algorithms for generating prefetches that are used for hardware DSM systems do not work well in Shasta because of the extremely large latencies and the higher overhead of issuing a prefetch (involves software protocol action). Our preliminary attempts at adding prefetches to LU and Volrend by hand have led to small improvements in performance (5%). We hope to have more results with prefetching soon.

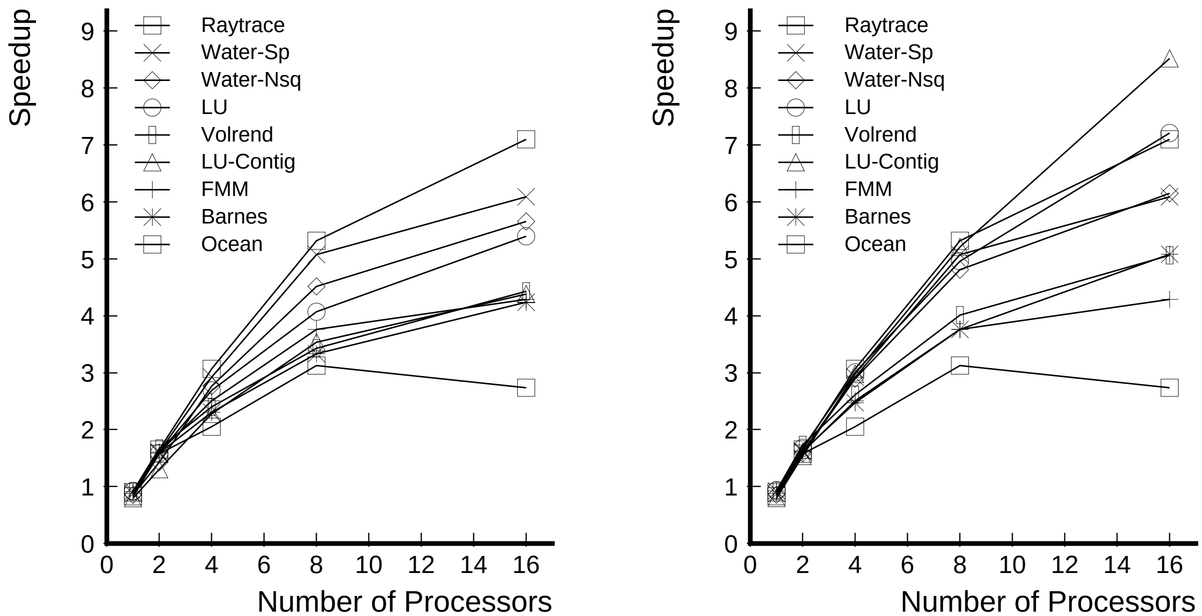

Figure 2: Speedups of SPLASH-2 applications running with 64-byte (left) and variable (right) block sizes.

All applications, except Ocean, achieve higher speedups as we use more processors. The performance drop in Ocean is primarily due to the fact that the Memory Channel bandwidth per processor drops as we go from 8 to 16 processors in our experimental setup. Raytrace achieves the highest speedup of 7.1 on 16 processors. The speedups are quite promising given the fast Alpha processors, the large communication latencies, and the relatively small application problem sizes.

To study the effects of variable coherence granularity, we made single-line changes to five of the applications to make the coherence granularity of one or a few of the main data structures greater than 64 bytes; the coherence granularity is a hint that can be specified at allocation time as a parameter to a modified malloc routine. The speedup curves on the right side of Figure 2 represent runs with the above coherence granularity hints used for a subset of the applications. Table 2 shows the affected data structures in each application along with the larger block size. We also show the change in speedups for 16-processor runs under Shasta when the larger granularity is used. Variable granularity improves performance by transferring data in larger units and reducing the number of misses on the main data structures. The most significant change occurs in LU-Contiguous, with the speedup almost doubling due to the larger block size (the number of read misses is reduced by over 25 times). Given the large potential gains from using appropriate communication granularities and the ease with which a programmer can experiment with this, support for variable granularity is an extremely effective mechanism for achieving higher performance.

Table 3 shows results for a subset of the applications running with slightly larger problem sizes. The table shows the larger input sizes along with other information similar to Table 1. In addition, we show the speedups achieved on 16 processors with the larger problem sizes. The miss check overheads are almost identical to the runs with smaller problem sizes; these overheads are not fundamentally dependent on the problem size. The speedups improve significantly, however. The speedup for Ocean, for example, improves by over two times with a doubling of the input parameter.

	selected data structure(s)	specified block size (bytes)	16-proc. speedup	
			default block size (64 bytes)	specified block size
Barnes	cell, leaf arrays	512	4.24	5.08
LU	matrix array	128	5.40	7.21
LU-Contig	matrix block	2048	4.38	8.52
Volrend	opacity, normal maps	1024	4.43	5.06
Water-Nsq	molecule array	2048	5.66	6.15
Water-Sp	molecules, boxes	varies	NA	6.09

Table 2: Effects of variable block size in Shasta.

	problem size	sequential time	with Shasta miss checks	miss check overhead	speedup (16 proc)
Barnes	64K particles	41.76s	45.38s	8.7%	6.74
LU	2048x2048 matrix	219.61s	236.4s	7.6%	9.43
LU-Contig	2048x2048 matrix	140.9s	176.5s	25.3%	10.47
Ocean	1026x1026 ocean	44.90s	53.90s	20.0%	5.70
Water-Nsq	4096 molecules	125.9s	147.3s	17.0%	9.71
Water-Sp	4096 molecules	15.94s	18.12s	13.7%	8.39

Table 3: Execution times for larger problem sizes (variable block sizes used where applicable).

4.4 Effect of Exploiting Release Consistency

This section analyzes the effect of the optimizations related to release consistency (described in Section 3.4), along with allowing multiple outstanding misses within a batch (described in Section 3.5).

Figure 3 presents the change in the execution time of 8- and 16-processor runs with a 64-byte block size for different levels of optimizations. For each application, the middle bar (labeled “B”) represents the execution time for the base runs reported in the previous section with the problem sizes specified in Table 1, and other times are normalized to this time. As was mentioned in the previous section, the base set of runs exploit all of the optimizations related to batching (multiple outstanding misses and merging of loads and stores to the same lines within a batch) and release consistency except that release operations are blocking (i.e., non-blocking stores, eager exclusive replies, and lockup-free optimizations are exploited). The height of the first bar for each application represents a conservative implementation that supports sequential consistency (labeled “SC”). These runs do not exploit the optimizations related to release consistency, and no overlap is allowed among misses within a batch (except for merging load and store misses to the same line). Finally, the third bar for each application represents the addition of non-blocking releases (labeled “NR”) to the set of optimizations used by the base set of runs. The figure shows that the optimizations used by the base set of runs help performance by as much as 10% relative to SC, with the gains more noticeable with fewer processors. However, the addition of the non-blocking release optimization does not visibly improve performance beyond the base set of runs and in some cases leads to slightly lower performance (which is why we do not use this optimization in our base set of runs).

Figure 3 also shows the breakdown of the execution time for each of the runs. Task time represents the time spent executing the application, including hardware cache misses. Task time also includes the time for executing the inline miss checks (and polls) and the code necessary to enter the protocol (such as saving registers). Read time and write time represent the stall time for read and write misses that are satisfied by other processors through the software protocol. Even though some of the runs exploit non-blocking stores, stores may still stall the processor if the number of pending requests exceeds a maximum threshold (100 requests per processor for these runs). For simplicity, our current implementation also stalls on a store if there are non-contiguous stores to a pending line. Synchronization time represents the stall time for application locks and barriers (includes both acquire and release times). Message time represents the time spent handling messages when the processor is not already stalled. Processors also handle messages while stalled on data or synchronization, but this time is hidden by the read, write, and synchronization times.

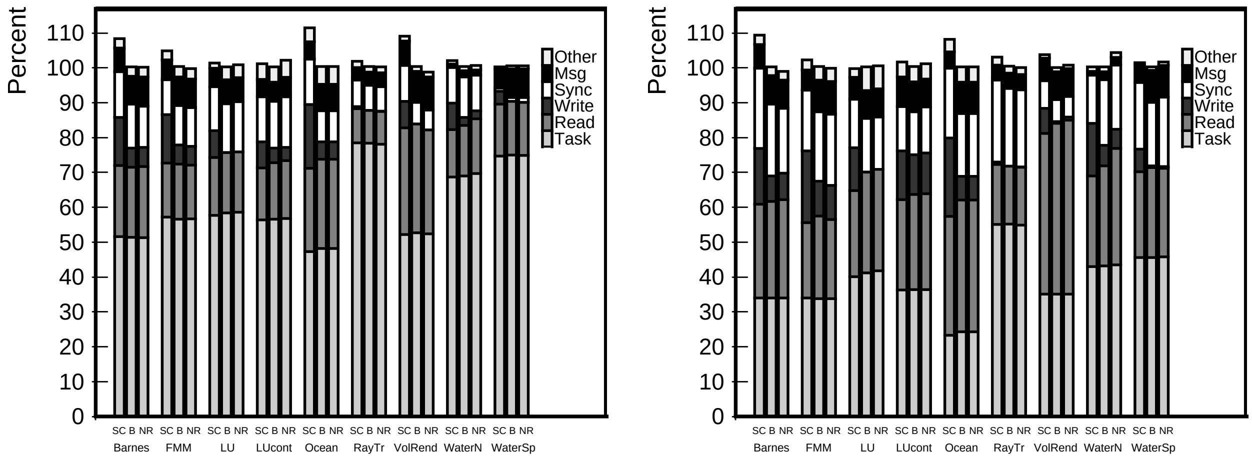

Figure 3: Effect of relaxed memory consistency for 8-processor (left) and 16-processor (right) runs with 64-byte block size.

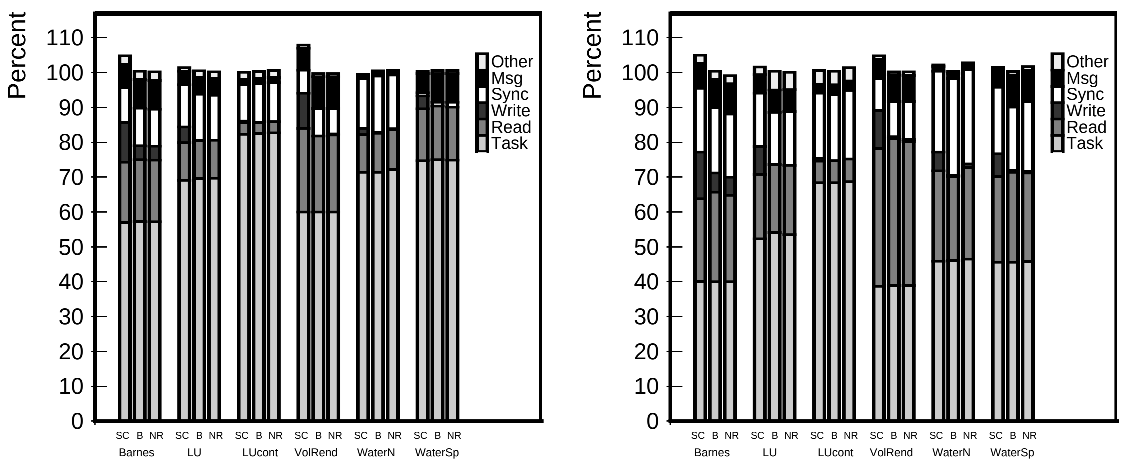

Figure 4: Effect of relaxed memory consistency for 8-processor (left) and 16-processor (right) runs with variable block size.

The “other” category includes miscellaneous overheads such as the overhead of dealing with non-blocking stores to pending lines. As expected, the 16-processor runs spend a higher percentage of their execution time in overhead categories compared to 8-processor runs.

The breakdowns in Figure 3 show that the optimizations used in the base set of runs are effective in significantly reducing and sometimes eliminating (for LU, Volrend, and Water-Sp) the write stall time.[5] Barnes and Ocean get a performance improvement of about 10% relative to SC at both 8 and 16 processors. Volrend and FMM also get visible improvements at 8 processors (5-10%), but the improvements are less at 16 processors.

The overall gains from the optimizations in the base set of runs relative to SC are much more limited than we initially expected. In most cases, the reduction in write stall time is accompanied by an increase in other overhead categories. Even though the processors do not directly stall for stores, the pending store requests still require servicing and can increase the time for other protocol operations. This leads to increases in read, synchronization, message

[5]LU-Contiguous gets only a small reduction in write stall time. Virtually all the stores are upgrade requests that are satisfied immediately by the local home but generate a single invalidation request which takes much longer to complete. This leads to stalls due to reaching the threshold of 100 pending requests outstanding per processor. While increasing this threshold leads to a reduction in the write stall time, there is no improvement in performance since the time simply shifts to other components of the execution time.

handling, or other miscellaneous overhead times. For example, at 16 processors, the write stall times for LU and Water-Sp are completely eliminated and the write times for FMM and Water-Nsq are approximately halved, yet there is little or no improvement in performance due to increases in other categories.

To better explain the above, we gathered more detailed data on how much time is spent by the processors to handle incoming protocol messages. As shown in Figure 3, the contribution of message handling time (Msg category) to the total execution time is less than 10% across the applications. Nevertheless, a large portion of the messages are handled while a processor is waiting for its own data and synchronization requests. Our more detailed data for 16-processor runs shows that for Barnes, FMM, Ocean, and Water-Nsq, the processors are busy handling incoming messages for an average of 30-35% of the time while they are waiting for data and synchronization (corresponds to the sum of read, write, and synchronization categories); for Volrend, the processors are busy for about 25% of this time and for the remaining applications, this percentage is around 17-20%. Therefore, the processors are heavily utilized while they "wait" for their own requests to complete. This explains why hiding the stall time for some operations may easily lead to higher stall times for other operations.

The addition of non-blocking releases to the base set of optimizations does not lead to any significant increase in performance and in fact leads to slightly lower performance in a few of the runs. The percentage of execution time spent on stalled releases for the base runs with 16 processors are as follows (based on our more detailed results): 3.8% for Water-Nsq, 1.5% for Barnes and FMM, 0.5% for Raytrace and Volrend, and negligible for the other applications. Barnes, FMM, and Raytrace show small gains from non-blocking releases. However, even though one would expect Water-Nsq to get the largest gain from this optimization, the performance actually suffers slightly. While non-blocking releases eliminate the release stalls in Water-Nsq, the time to acquire locks almost doubles[6] and read latencies increase slightly, leading to a higher overall execution time.

Figure 4 presents similar data to Figure 3 for runs with variable block sizes (specified in Table 2) for the subset of applications that exploit this feature. As expected, the breakdowns indicate a higher efficiency (i.e., higher percentage for Task time) compared to the runs with a 64-byte block size. Nevertheless, the trends are very similar to those observed in Figure 3. Optimizations that exploit relaxed models are still effective in significantly reducing the write stall times, with Barnes and Volrend showing a 5-10% performance improvements with the larger block sizes.

We did a number of additional experiments to further isolate the effect of various optimizations used in the base set of runs. The first set of experiments allowed for the overlap of multiple misses in batches relative to the SC runs. This led to virtually no improvement in performance, except for Volrend which achieved a 5% improvement relative to SC when using 8 processors and a 64-byte block size. This is mainly because there is rarely more than a single shared miss at a each batch. The second set of experiments added the eager exclusive reply optimization whereby the reply data to a read exclusive request is used by the requesting processor before all invalidations are acknowledged; in this case, the only missing optimization compared to the base set of runs is non-blocking stores. Again, this additional optimization did not improve performance in most of the cases. Ocean achieved a 5% gain with a 64-byte block size at both 8 and 16 processors, and Barnes achieved similar gain only at 16 processors. Therefore, as expected, much of the performance difference between SC and the base runs can be attributed to the non-blocking store optimization.

4.5 Effect of Upgrades and Dirty Sharing

We analyze the effect of supporting exclusive (or upgrade) requests and supporting dirty sharing in this section. Figure 5 presents a breakdown of the execution times for 8- and 16-processor runs with a 64-byte block size. Again, the middle bar (labeled "B") for each application represents the execution time for the base runs reported in Section 4.3, and other times are normalized to this time. The base set of runs use upgrade requests (i.e., no data is fetched on a store if the processor already has a shared copy), and do not use sharing writeback messages (i.e., home is not updated on 3-hop read operations). The first bar for each application represents an implementation that does not support upgrade messages (labeled "NU"). Therefore, a processor generates a read-exclusive request whether or not there is a local shared copy of the line. The third bar for each application represents the addition of sharing writeback messages to the base set of experiments (labeled "WB"). Figure 6 presents similar data except for runs with variable block sizes.

[6] This is because some of the releases are on the critical path, and while a non-blocking release removes the stall time from the releasing processor, it does not lead to a faster time for releasing a waiting processor.

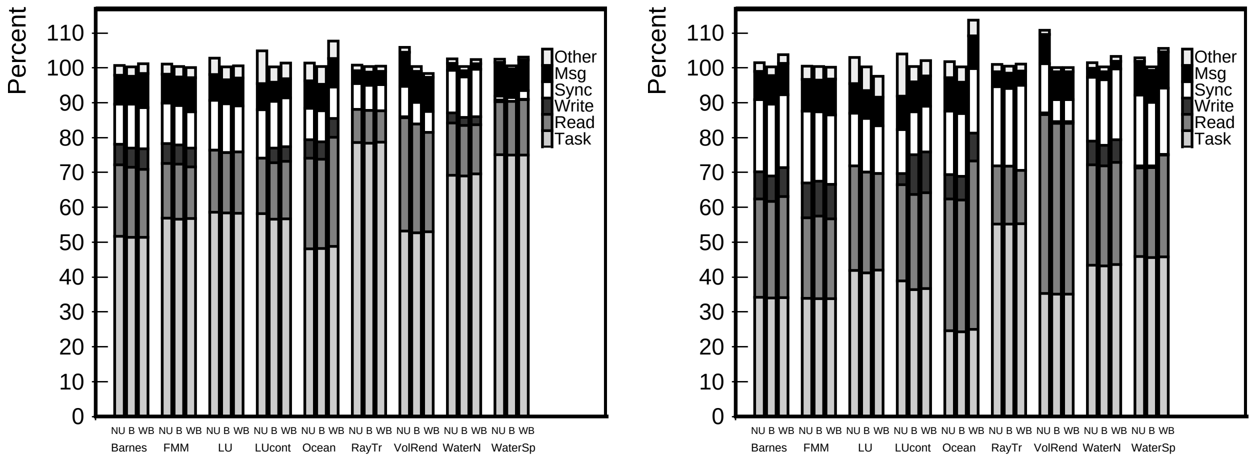

Figure 5: Effect of upgrades and sharing writebacks for 8-processor (left) and 16-processor (right) runs with 64-byte block size.

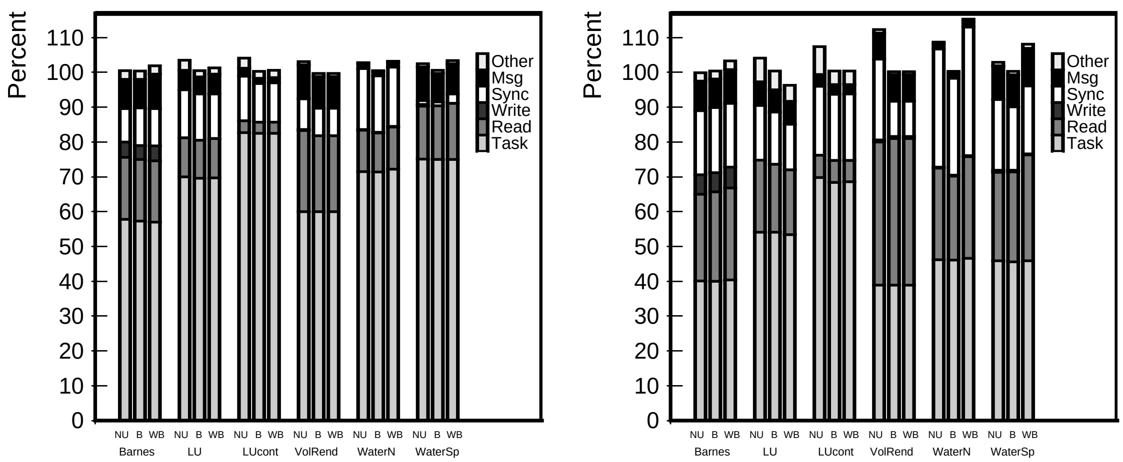

Figure 6: Effect of upgrades and sharing writebacks for 8-processor (left) and 16-processor (right) runs with variable block size.

The results show that support for upgrade messages is important for a number of the applications. For example, Volrend achieves above a 10% improvement in performance due to upgrade messages at 16 processors for both 64-byte and variable block sizes. On the other hand, sharing writeback messages typically hurt performance (which is why we do not use it in the base set of runs). The only application that achieves visible gains from sharing writebacks is LU at 16 processors (for both the 64-byte and the larger block size runs). This is due to the fact that several processors read the data produced by another processor. On the other hand, applications such as Ocean that have single-producer/single-consumer sharing patterns are hurt by the additional messages generated by the writebacks. Larger block sizes can sometimes exacerbate the cost of the writebacks, as is shown for Water-Nsq where the performance degrades by over 15%. Therefore, supporting a dirty-sharing protocol is important for achieving higher performance in Shasta.

4.6 Effect of Migratory Optimizations

Figure 7 shows the effect of migratory optimizations described in Section 3.6 for 8- and 16-processor runs with a 64-byte block size. Again, the first bar (labeled "B") for each application represents the execution time for the base

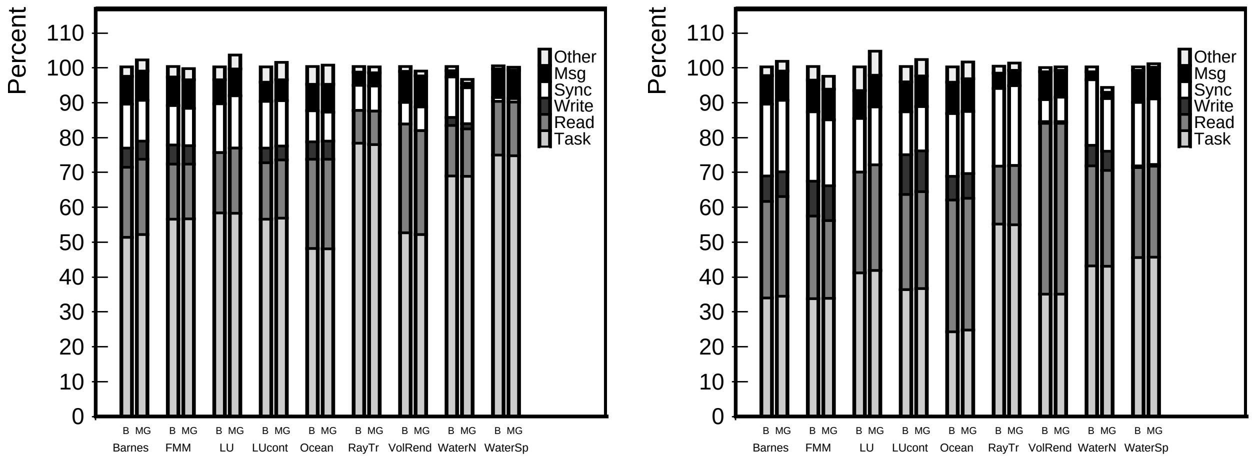

Figure 7: Effect of migratory optimizations for 8-processor (left) and 16-processor (right) runs with 64-byte block size.

runs reported in Section 4.3, and other times are normalized to this time. The second bar for each application represents runs with the migratory optimizations (labeled "MG"). Figure 8 presents similar data except for runs with variable block sizes. We used a threshold of 2 times for observing the migratory pattern before switching a line to migratory conversion, and used a threshold of 5 reversions after which we stop considering the line for conversion.

The results for migratory optimizations are quite disappointing. The optimization either does not provide an improvement or reduces performance slightly in the majority of cases. In fact, the performance degradations would be much worse without the sophisticated revert mechanism and hysteresis built into our protocol. The primary reason for the poor performance is the fact that migratory patterns are either not present at the granularity of 64-byte or larger block sizes or the pattern is unstable (i.e., reverts back to non-migratory behavior).

At 16 processors and 64-byte block sizes, LU, LU-Contiguous, Ocean, and Water-Sp detect virtually no migratory sharing, Raytrace and Volrend detect a few stable patterns, and Barnes and FMM detect a few patterns that are unstable. The only application that is successful in detecting a large number of stable patterns is Water-Nsq. In fact, the number of upgrade misses is reduced by over 90% in this application. Hence, Water-Nsq is the only application that shows visible gains from this optimization. At the larger block sizes, even Water-Nsq ends up having fewer and more unstable patterns, therefore we actually see a slight loss in performance in Figure 8.

Aside from the lack of stable migratory patterns, there are several other factors that reduce the potential gains from migratory optimizations in Shasta. First, the use of upgrade messages reduces the cost of store misses that may be eliminated. Second, exploiting release consistency is effective in hiding the latency of the upgrades. Finally, the batching optimization also leads to the merging of load and store misses to the same line within a single batch.

4.7 Summary of Results

Overall, support for variable granularity communication is by far the most important optimization in Shasta. Support for upgrade messages and a dirty-sharing protocol are also important for achieving higher performance. Exploiting release consistency provides smaller performance gains than we expected due to the fact that processors are busy handling protocol messages while they are waiting for their own requests to complete. Finally, migratory optimizations turn out not to be useful in the context of Shasta.

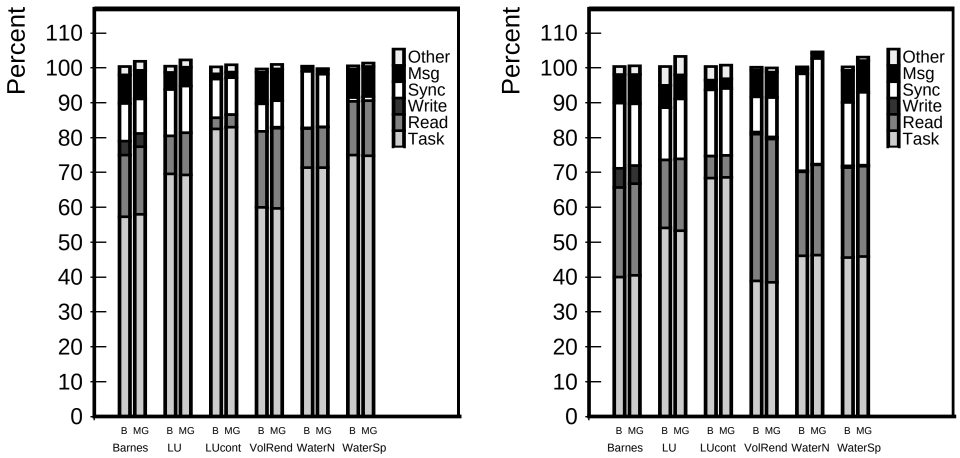

Figure 8: Effect of migratory optimizations for 8-processor (left) and 16-processor (right) runs with variable block size.

5 Discussion and Related Work

Shasta's basic approach is derived from the Blizzard-S work [21]. However, we have substantially extended the previous work in this area by developing several techniques for reducing the otherwise excessive access control overheads. We have also developed an efficient protocol that provides support for maintaining coherence at variable granularities within a single application. Finally, we have explored the use of relaxed memory models in the context of software protocols that can support coherence at a fine granularity. In fact, this is the first paper to analyze the effect of such protocol optimizations in the context of a fine-grain software DSM system. In a separate paper, we describe a major extension to the Shasta protocol that exploits SMP nodes by allowing processors to efficiently share memory within the same SMP [18].

Object- or region-based DSM systems [1, 3, 12, 17, 20] communicate data at the object level and therefore support coherence at multiple granularities, but these systems require explicit programmer intervention to partition the application data into objects and to identify when objects are accessed through annotations. Midway also allows different regions of memory to have different granularities for detecting writes. Even though a finer granularity of write detection can reduce the amount of communicated data, the access and coherence granularity is still at an object or page level (depending on the consistency model). Similarly, some page-based systems (e.g., Treadmarks [14]) reduce the required bandwidth by only communicating the differences between copies, but the coherence granularity is still a page. Page-based DSM systems implemented on a cluster of shared-memory multiprocessors, such as MGS [25] and SoftFLASH [7], naturally support two coherence granularities – the line size of the multiprocessor hardware and the size of the virtual memory page. However, neither of these granularities can be changed.

There has been a lot of research on exploiting and evaluating the benefits of relaxed memory consistency models in the context of both hardware and software DSM systems. The studies involving software DSM systems have all focused on page-based systems [2, 4, 5, 11, 13, 26]. The use of relaxed models is critical for alleviating false sharing problems that arise due to the large coherence granularity in such systems. Therefore, performance gains from models such as release consistency relative to sequential consistency are quite large in this context. In contrast, Shasta alleviates false sharing by supporting coherence at a fine granularity, and therefore does not depend on relaxed models for this purpose. Because of this, the coherence protocol used in Shasta more closely resembles that of hardware DSM systems. However, the performance gains from exploiting a relaxed model are much more limited in Shasta compared to hardware systems [8], even though communication latencies are much larger. The primary reason for this is that the processors are utilized for handling protocol messages while they wait for data and synchronization, thus making it more difficult to improve performance through latency hiding.

Cox and Fowler [6], and Stenstrom et *al.* [23], independently proposed the idea of optimizing the transfer of migratory data and evaluated the performance of this optimization (through simulation) for a small number of applications in the context of hardware DSM systems. Both studies focus on a small block size of 16 bytes. They both observe stable migratory behavior at this granularity. Stenstrom et *al.* also describe a mechanism for reverting from migratory conversion, but show that this mechanism is not required for good performance. Cox and Fowler provide some simulation results for larger block sizes and notice a degradation of migratory patterns at larger sizes. Our experience with Shasta shows that migratory patterns are indeed unstable or not present in systems with block sizes of 64 bytes or larger, thus limiting the gains from this optimization. Similarly, sophisticated mechanisms are required for reverting from migratory conversion in order to limit performance degradations when the sharing pattern is unstable.

6 Conclusion

Shasta is a software distributed shared memory system that supports fine-grain access to shared memory by inserting code before loads and stores in an application that checks the state of the shared data being accessed. We have implemented an efficient cache coherence protocol that incorporates a number of optimizations, some of which are common in hardware shared memory systems. This protocol is fully functional and runs on a prototype cluster of Alpha processors connected through Digital's Memory Channel network.

Since Shasta supports shared memory entirely in software, it provides considerable flexibility in managing coherence granularity and applying protocol optimizations. Our detailed parallel performance results illustrate the benefits of this flexibility by isolating the effects of various protocol optimizations. The ability to support multiple coherence granularities within a single application is by far the most unique and important feature of Shasta, leading to performance improvements of as high as two times. Other protocol features, such as support for dirty sharing and exclusive (or upgrade) messages, are also shown to be important for achieving high performance. Techniques for hiding latency, such as exploiting relaxed memory consistency models, lead to more limited performance gains. This is primarily because processors are often utilized for handling protocol messages while they wait for their requests to complete, making gains from latency hiding less likely. Finally, optimizations based on detecting migratory sharing patterns turn out not to be promising in the context of Shasta.

References

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[2] J. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *Proceedings of the Second ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON 1993*, pages 528–537, Mar. 1993.

[4] 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*, pages 152–164, Oct. 1991.

[5] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[6] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[7] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, Oct. 1996.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[10] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, pages 12–18, Feb. 1996.

[11] L. Iftode, C. Dubnicki, E. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proceedings of the 2nd Symposium on High-Performance Computer Architecture*, Feb. 1996.

[12] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, pages 213–228, Dec. 1995.

[13] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, May 1992.

[14] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 94–105, May 1990.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[17] R. S. Nikhil. Cid: a Parallel, "Shared-memory" C for Distributed-memory Machines. In *Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 376–390, Aug. 1994.

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

[19] 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 Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Oct. 1996.

[20] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 101–114, Nov. 1994.

[21] 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 Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, Oct. 1994.

[22] J. P. Singh, W. D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[23] P. Stenstrom, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[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 International Symposium on Computer Architecture*, pages 24–36, June 1995.

[25] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–56, May 1996.

[26] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996.

WRL Research Reports

‘‘Titan System Manual.’’ **Michael J. K. Nielsen.** WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’ **David W. Wall.** WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’ **William R. Hamburgen.** WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’ **David W. Wall and Michael L. Powell.** WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’ **Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.** WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’ **Christopher A. Kent, Jeffrey C. Mogul.** WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’ **Christopher A. Kent.** WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’ **David W. Wall.** WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’ **Paul J. Asente.** WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’ **Brian K. Reid.** WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’ **Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.** WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’ **Jeremy Dion.** WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’ **Joel F. Bartlett.** WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’ **Jeffrey C. Mogul.** WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’ **David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.** WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’ **Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.** WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’ **Joel F. Bartlett.** WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’ **Silvio Turrini.** WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’ **William R. Hamburgen.** WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’ **Jeffrey C. Mogul.** WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’ **V. Srinivasan and Jeffrey C. Mogul.** WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’ **Norman P. Jouppi and David W. Wall.** WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’ **Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.** WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’ **Norman P. Jouppi.** WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’ **Norman P. Jouppi.** WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’ **Norman P. Jouppi and Jeffrey Y. F. Tang.** WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’ **Norman P. Jouppi.** WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’ **Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.** WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’ **David W. Wall.** WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’ **Jeffrey Y.F. Tang and J. Leon Yang.** WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’ **Tracy Larrabee.** WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’ **Tracy Larrabee.** WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’ **Michael N. Nelson.** WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’ **John S. Fitch.** WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’ **Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.** WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’ **Joel McCormack.** WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’ **J. Bradley Chen, Anita Borg, Norman P. Jouppi.** WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’ **Don Stark.** WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’ **David Boggs.** WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’ **Scott McFarling.** WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’ **Joel Bartlett.** WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’ **G. May Yip.** WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’ **William R. Hamburgen.** WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’ **David W. Wall.** WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’ **Jeffrey C. Mogul.** WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’ **Norman P. Jouppi.** WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’ **William R. Hamburgen, John S. Fitch.** WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’ **David W. Wall.** WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’ **Russell Kao.** WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’ **Amitabh Srivastava and David W. Wall.** WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’ **Joel McCormack & Bob McNamara.** WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’ **Norman P. Jouppi & Steven J.E. Wilton.** WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’ **Amitabh Srivastava.** WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’ **Steven J.E. Wilton and Norman P. Jouppi.** WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’ **Alberto Makino, William R. Hamburgen, John S. Fitch.** WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’ **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.** WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’ **Amitabh Srivastava, David W. Wall.** WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’ **Amitabh Srivastava, Alan Eustace.** WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’ **Keith I. Farkas, Norman P. Jouppi.** WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’ **Jeffrey C. Mogul.** WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo, Herve Touati.** WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’ **J. Bradley Chen, David W. Wall, and Anita Borg.** WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’ **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.** WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’ **Keith I. Farkas, Norman P. Jouppi, and Paul Chow.** WRL Research Report 94/8, December 1994.

''Drip: A Schematic Drawing Interpreter.'' **Ramsey W. Haddad.** WRL Research Report 95/1, March 1995.

''Recursive Layout Generation.'' **Louis M. Monier, Jeremy Dion.** WRL Research Report 95/2, March 1995.

''Contour: A Tile-based Gridless Router.'' **Jeremy Dion, Louis M. Monier.** WRL Research Report 95/3, March 1995.

''The Case for Persistent-Connection HTTP.'' **Jeffrey C. Mogul.** WRL Research Report 95/4, May 1995.

''Network Behavior of a Busy Web Server and its Clients.'' **Jeffrey C. Mogul.** WRL Research Report 95/5, October 1995.

''The Predictability of Branches in Libraries.'' **Brad Calder, Dirk Grunwald, and Amitabh Srivastava.** WRL Research Report 95/6, October 1995.

''Shared Memory Consistency Models: A Tutorial.'' **Sarita V. Adve, Kourosh Gharachorloo.** WRL Research Report 95/7, September 1995.

''Eliminating Receive Livelock in an Interrupt-driven Kernel.'' **Jeffrey C. Mogul and K. K. Ramakrishnan.** WRL Research Report 95/8, December 1995.

''Memory Consistency Models for Shared-Memory Multiprocessors.'' **Kourosh Gharachorloo.** WRL Research Report 95/9, December 1995.

''Register File Design Considerations in Dynamically Scheduled Processors.'' **Keith I. Farkas, Norman P. Jouppi, Paul Chow.** WRL Research Report 95/10, November 1995.

''Optimization in Permutation Spaces.'' **Silvio Turrini.** WRL Research Report 96/1, November 1996.

''Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.'' **Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath.** WRL Research Report 96/2, November 1996.

''Efficient Procedure Mapping using Cache Line Coloring.'' **Amir H. Hashemi, David R. Kaeli, and Brad Calder.** WRL Research Report 96/3, October 1996.

''Optimizations and Placement with the Genetic Workbench.'' **Silvio Turrini.** WRL Research Report 96/4, November 1996.

''Performance of the Shasta Distributed Shared Memory Protocol.'' **Daniel J. Scales and Kourosh Gharachorloo.** WRL Research Report 97/2, February 1997.

''Fine-Grain Software Distributed Shared Memory on SMP Clusters.'' **Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal.** WRL Research Report 97/3, February 1997.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’ **Brian K. Reid and Christopher A. Kent.** WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’ **Christopher A. Kent.** WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’ **Joel McCormack.** WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?.’’ **John Ousterhout.** WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’ **Joel F. Bartlett.** WRL Technical Note TN-12, October 1989.

‘‘Characterization of Organic Illumination Systems.’’ **Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.

‘‘Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.’’ **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’ **Jeffrey C. Mogul and Anita Borg.** WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’ **Aaron Goldberg and John Hennessy.** WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’ **David W. Wall.** WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion.’’ **Scott McFarling.** WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach.’’ **John S. Fitch.** WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter.’’ **David Boggs.** WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package.’’ **Patrick D. Boyle.** WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics.’’ **Joel F. Bartlett.** WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0.’’ **Jeremy Dion & Louis Monier.** WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture.’’ **Amitabh Srivastava and David W. Wall.** WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors.’’ **Scott McFarling.** WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo and Herve Touati.** WRL Technical Note TN-37, June 1993.

‘‘Piecewise Linear Models for Rsim.’’ **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.

''Speculative Execution and Instruction-Level Parallelism.'' **David W. Wall.** WRL Technical Note TN-42, March 1994.

''Ramonamap - An Example of Graphical Groupware.'' **Joel F. Bartlett.** WRL Technical Note TN-43, December 1994.

''ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.'' **Alan Eustace and Amitabh Srivastava.** WRL Technical Note TN-44, July 1994.

''Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS.'' **Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.** WRL Technical Note TN-45, March 1994.

''Experience with a Wireless World Wide Web Client.'' **Joel F. Bartlett.** WRL Technical Note TN-46, March 1995.

''I/O Component Characterization for I/O Cache Designs.'' **Kathy J. Richardson.** WRL Technical Note TN-47, April 1995.

''Attribute caches.'' **Kathy J. Richardson, Michael J. Flynn.** WRL Technical Note TN-48, April 1995.

''Operating Systems Support for Busy Internet Servers.'' **Jeffrey C. Mogul.** WRL Technical Note TN-49, May 1995.

''The Predictability of Libraries.'' **Brad Calder, Dirk Grunwald, Amitabh Srivastava.** WRL Technical Note TN-50, July 1995.

WRL Research Reports and Technical Notes are available on the World Wide Web, from `http://www.research.digital.com/wrl/techreports/index.html`.