Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance

Princeton university Technical Report TR-517-96.

Submitted for publication. Please do not distribute.

Angelos Bilas, Liviu Iftode, David Martin, and Jaswinder Pal Singh
Department of Computer Science, Princeton University, Princeton, NJ 08544

Abstract

As the workstation market moves form single processor to small-scale shared memory multiprocessors, it is very attractive to construct larger-scale multiprocessors by connecting widely available symmetric multiprocessors (SMPs) in a less tightly coupled way. Using a shared virtual memory (SVM) layer for this purpose preserves the shared memory programming abstraction across nodes. We explore the feasibility and performance implications of one such approach by extending the AURC (Automatic Update Release Consistency) protocol, used in the SHRIMP multicomputer, to connect hardware-coherent SMPs rather than uniprocessors. We describe the extended AURC protocol, and compare its performance with both the AURC uniprocessor node case as well as with an all-software Lazy Release Consistency (LRC) protocol extended for SMPs. We present results based on detailed simulations of two protocols (AURC and LRC) and two architectural configurations of a system with 16 processors: one with one processor per node (16 nodes) and one with four processors per node (4 nodes).

We find that, unless the bandwidth of the network interface is increased, the network interface becomes the bottleneck in a clustered architecture especially for AURC. While a LRC protocol can benefit from the reduction in per processor communication in a clustered architecture, the write-through traffic in AURC increases significantly the communication demands per network interface. This causes more traffic contention and either prevents the performance of AURC from improving under SMP or hurts it severely for applications with significant communication requirements. Thus, while AURC performs better than LRC, for applications with high communication needs, the reverse may be true in clustered architectures. Among possible solutions, two are investigated in the paper: protocol changes and bandwidth increases. Further work is clearly needed on the systems and application sides to evaluate whether AURC can be extended for multiprocessor node systems.

1 Introduction

While hardware cache-coherent distributed shared memory machines have been shown to perform well at least on a moderate scale, they are expensive and complex to build. Many research efforts have recently investigated supporting a coherent shared address space with less hardware support, ranging from less customized and integrated coherence controllers [12] to supporting shared virtual memory (SVM) at page level through the operating system [9, 6]. For example, the SHRIMP multicomputer [1] takes the approach of using commodity PCs and attaching a hardware network interface to them. This network interface consists of two parts: an I/O card that plugs into the I/O bus and is used for communication with the outside world, and a second simple card that snoops the memory bus. The I/O card uses user level DMA to transfer data between the network and the memory. This mechanism is called Deliberate Update. The snoop card is used to support virtual memory mapped communication. In particular, writes to local pages for which a remote mapping has been established are automatically propagated to the remote pages. The Automatic Update data bypass the I/O bus and are sent to the I/O card through a private bus. The Automatic Update mechanism is transparent to the user. It is possible to use such minimal hardware support to build an SVM layer that outperforms all-software SVM implementations [6, 7]. One such protocol, called AURC (Automatic Update Release Consistency), uses lazy release consistency [10] together with the automatic update mechanism to implement a multiple writer protocol [6]. A similar approach using a directory-based scheme has been taken on other systems that provide automatic update support [11].

Recently, small-scale shared-memory SMPs have become increasingly widespread. Inexpensive SMPs based on Intel PC processors are on the market, and SMPs from other vendors are increasingly popular. Given this development, it is natural to examine whether the SVM mechanisms developed for uniprocessor nodes extend well to using an SMP as the base node, and how these mechanisms can be improved for this task. In this environment, the bus coherence protocol operates at cache line level within an SMP, and the higher-overhead SVM mechanism provides communication and coherence at page granularity across SMPs. Another view of this approach is that the less efficient SVM is used not as the basic mechanism with which to build multiprocessors out of uniprocessors, but as a mechanism to extend available small-scale machines to build larger machines while preserving the same desirable programming abstraction. The trends in the marketplace make this two-level approach attractive.

The first question to be answered is, how will using k, c-processor SMPs, connected this way, compare to using SVM across $k * c$ uniprocessor nodes. Clustering processors together using a faster and finer-grained communication mechanism has some obvious advantages; namely prefetching, cache-to-cache sharing and overlapping working sets [4]. The hope is that for many applications a significant fraction of the interprocessor communication may be contained within each SMP node. This reduces the amount of expensive (high latency and overhead) cross-node SVM communication needed during the application's execution. However, it unfortunately *increases* the bandwidth (i.e. communication per unit time) demands on the node-to-network interface. This is because the combined computation power within the SMP node typically increases much faster with cluster size c (linearly) than the degree to which the per processor cross-node communication volume is reduced. This means that depending on the constants, the node-to-network bandwidth may become a bottleneck if it is not increased considerably when going from a uniprocessor to an SMP node.

A second question to be answered is whether the communication bottlenecks exposed by the SMP configuration are inherent to SVM or they stem from the write-through nature of the automatic update mechanism used in AURC. To investigate this question, we have also simulated an all-software LRC protocol similar to the one used in TreadMarks, slightly adapted to take advantage of the multiprocessor configuration of nodes.

In this context, the contribution of this paper is twofold. First,it describes how to extend the AURC protocol to use hardware-coherent SMPs as the basic nodes, and examines the issues in this process. Then, it examines the performance of a few applications with different communication requirements for which the performance on uniprocessor-node LRC and AURC systems is well understood [7]. It also compares two extended protocols (MP-AURC and MP-LRC) with different communication traffic requirements. We find that clustering in SMP nodes does indeed reduce the amount of SVM communication per *processor* (e.g. page faults), but causes increased aggregate bandwidth requirements per network interface. This affects AURC substantially more than LRC because of the contention effects induced through automatic update. The performance of MP-AURC can be improved by reducing the automatic update traffic and/or increasing the node-to-network bandwidth. Overall the AURC family of protocols performs better than the LRC protocols. However for applications with high communication needs LRC may perform better than AURC in the multiprocessor node case, but still worse than AURC in the uniprocessor case, which remains the best performing system overall.

The rest of the paper is organized as follows. In Section 2 we describe the AURC protocol for a system with one processor per node. Section 3 discusses the extensions to the protocol for SMPs and the lower-level issues that must be resolved. The simulator we use is described in Section 4 along with the positions we took on lower-level issues. Section 5 presents the applications we used in the simulations. Performance results and analysis are presented in Section 6. Finally, Section 7 summarizes and concludes the paper.

2 Automatic Update and AURC

The Automatic update mechanism can be viewed as the propagation of local writes to remote memory. In that sense the remote memory is a copy of the local memory updated with a write-through mechanism. Thus Automatic Update allows selective writes to be performed twice (doubled) both on a local memory page as well as on a remote memory page, assuming a mapping was previously set between the source and the destination. As mentioned above Automatic update is implemented by having the network interface snoop all write traffic on the memory bus. Pages with outgoing automatic update mappings use write-through caching, so that every CPU write to such a page causes a transaction on the memory bus. On seeing a write, the network interface hardware checks to see whether the page being written has an automatic update mapping; if it does, the network interface automatically creates a network packet and sends it to the correct destination. The AU feature provides useful support for a shared virtual memory implementation. Automatic Update Release Consistency (AURC) [6] is a protocol designed to support a lazy release consistency (LRC) model in this environment.

2.1 The AURC Protocol

The novel approach in AURC is to use automatic update mappings to merge updates from multiple writers on the same page into a unique copy of that page at the *home* node's memory (see Figure 1). This scheme replaces more expensive all-software solutions used in

LRC implementations for write detection like "diff" computation [9] or software dirty bit schemes [15]. While the home's copy is updated exclusively through fine-grain automatic updates, a non-home node obtains an up-to-date copy of the page when needed through a page fetch request sent to the home following a page fault due to a coherence invalidation. The coherence actions are performed entirely in software at synchronization time using vector time-stamps [10] to ensure a correct ordering of events according to the LRC model definitions.

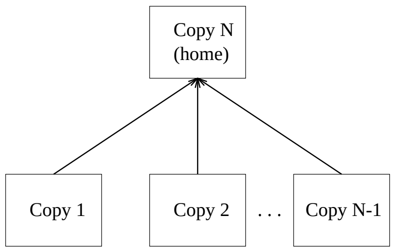

Figure 1: Automatic Update Mappings for N copies.

Pages shared by only two nodes can be treated specially. In this case, coherence can be maintained entirely by the hardware with reciprocal AU mappings between the two copies (see Figure 2). This special case protocol called Copyset-2 is part of AURC and is initially chosen for each shared page until more than two copies are created. At this time, AURC switches to the general scheme described above (Figure 1) which is called Copyset-N. By combining these two AU-based schemes, AURC can reduce the communication cost by avoiding the page faults for pages which are shared by only two nodes. The price for the reduced number of page faults is an extra automatic-update traffic.

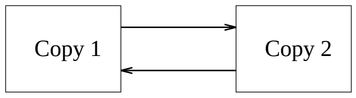

Figure 2: Automatic Update Mappings for 2 copies.

3 Clustered Architecture and Protocols

The reference architecture we use for our measurements is a shared virtual memory system consisting of a number of SMP clusters interconnected by a high-speed network. The clusters are bus-based SMPs each containing a number of processors, a memory module and an I/O interface. The memory bus is a split transaction bus. The network interface implements memory mapped communication between clusters and consists of two components: a user-level DMA engine connected to the I/O bus and an automatic-update snooper connected to the memory bus.

Shared memory coherence is supported at two levels. Intra-cluster coherence is maintained at cache-line level by a standard snoopy protocol. The inter-cluster coherence is implemented at page granularity in software using either MP-AURC, a shared virtual memory protocol that is an extension of AURC [6] or MP-LRC a simple extension of the LRC protocol for SMPs. In the following paragraphs we describe these extensions.

3.1 Inter-cluster Coherence

To a large degree, the extended MP-AURC and MP-LRC protocols are similar to the simple ones (AURC and LRC).

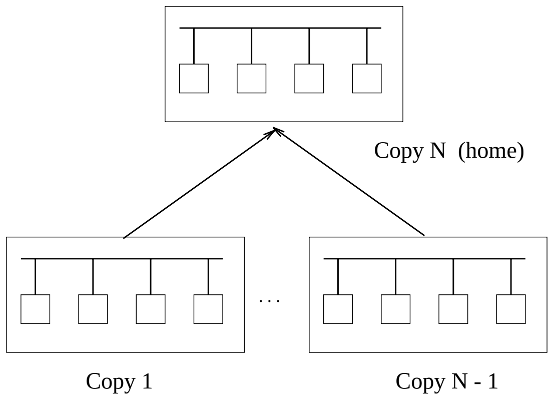

Figure 3: Automatic Update Mappings for N copies in MP-AURC.

To detect and collect the updates from multiple write clusters, MP-AURC establishes automatic update mappings from non-home clusters to the home-cluster of each shared page (Figure 3). Following these mappings, all local writes performed by any processor on a local copy are forwarded to the home cluster and performed at the cluster copy. Since the home page is updated exclusively through automatic updates, all processors of the home cluster can benefit from this through the intra-cluster coherence mechanism (i.e. none of them will have to page-fault for that page since it will be up to date when needed). For the non-home clusters, the fetch request is issued by the first processor which needs to access the page following an invalidation. In MP-AURC, the Copyset-2 scheme can lead to a full AU-based coherence between up to two nodes as illustrated in Figure 4) and may substantially increase the automatic update traffic.

Diff computation in MP-LRC is performed eagerly by any processor in the node at synchronization points as in LRC. Although previous work [3] has shown that lazy diffing performs slightly better for LRC, we chose eager diffing for simplicity, since we intended to use MP-LRC only as a base for performance comparison

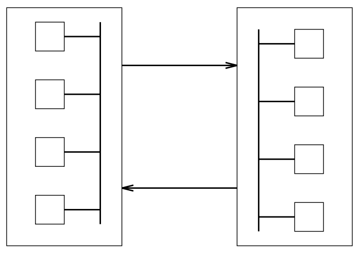

Figure 4: Automatic Update Mappings for 2 copies in MP-AURC.

with MP-AURC. For the same reason we didn't implement a scheduling policy which chooses idle processors in the node for diff computation as suggested by [8]. This work has shown that most of the time there is at least an idle processor which can take over the protocol work for the entire cluster. It is not straightforward though how the idle processors can be identified and how the protocol tasks should be scheduled.

Besides the similarities between the simple and the extended protocols, there are significant differences as well. In the following paragraphs we describe these differences.

3.1.1 Per-Node Shared Control Data

Shared pages in each node are owned by the node and not a specific processor in the node. Thus the version of the shared pages (expressed as a time-stamp vector) is shared by all processors in the same node. Write notices which contain the coherence information, are also maintained per node. Whenever a processor in the node reaches a synchronization point, a new interval for that node is created and the corresponding write notices are attached to the new interval.

3.1.2 Intra-cluster Acquire

When a lock is transferred from one processor to another processor inside the same cluster, no new interval is created because the intra-cluster coherence is supported in hardware. However write-notices from previous inter-cluster acquires are inspected to produce the required coherence invalidations at the acquire processor.

3.1.3 Servicing Remote Fetch Requests

A remote fetch request arriving at an SMP cluster can be satisfied by any processor in the cluster, since they all share the physical page and the protocol data structures. For MP-AURC as well as for MP-LRC in the absence of lazy diff computation, this overhead is small. It involves mainly starting two DMA transfers for each remote fetch request. This is why both nominating a fixed processor in the cluster (that is also a compute processor participating in the application) for protocol processing, or using a round-robin assignment in the node for remote fetch requests are reasonable options. We use the former for simplicity and our results show that the protocol overhead is negligible.

3.1.4 Eager vs. Lazy TLB Invalidation

In MP-LRC updating a page is done most of the time by applying diffs and not by page transfers. An important difference between diffs and page fetches is that diffs overwrite only portions of the page that have been updated whereas page fetches overwrite the whole page. In this case there is no need to forbid other processors from accessing the page while it is updated. When a full page is fetched, as is always the case in MP-AURC, the situation is different. After the page has been sent from the home node, all processors in the requesting node should stop writing this page, since the data will be overwritten by the older version of the fetched page. To prevent processors in the node from losing data, all processors in the node must flush their write buffers and invalidate the corresponding TLB entry before a page transfer request can be issued by a faulting processor. The TLB entry invalidations can be performed either eagerly or lazily. The eager approach invalidates the corresponding entry in all TLBs not at fetch time but at the earlier time of an acquire event performed by any processor in the cluster. The drawback of this solution is that unnecessary page invalidations and page misses can be caused to other processors in the cluster through false sharing effects. A lazy approach is to invalidate the TLBs just before the page fetch request itself is sent to the home, in which case the faulting processor may have to wait for write buffer flushes and TLB invalidations at other processors in the cluster to complete. In the current simulations we used lazy TLB invalidations.

3.1.5 Zero-action TLB Misses

In the SMP protocols not every TLB miss requires a remote fetch. This is because once a page or a diff is transferred on behalf of one processor, it can be used by all other processors in the same cluster as long as it satisfies the coherence requirements. In this case, remote transfers are avoided but the TLB misses still occur, since the TLB invalidations are performed on a per-processor basis.

3.1.6 Hierarchical Barriers

Barriers are expensive synchronization events in SVM systems if implemented using a central manager. Their cost can be alleviated in by using a hierarchical approach [2]. After all processors within a cluster have arrived at the barrier, one of them collects the write-notices from all four and sends them to the barrier manager (rather than have each processor send a message). Once all clusters have sent this message, the barrier manager replies to each cluster with sets of missing write-notices which are then are performed by each processor in the cluster.

4 Simulation Environment

We use an execution-driven simulator based on the TangoLite reference generator [5]. We couple TangoLite to a memory system simulator for MP-AURC as well as for AURC. We start with a set of aggressive next-generation parameters for the system components, and then in some cases vary parameters to understand their effects.

The basic model assumes a cluster of k SMPs connected with a high speed interconnect. Each node in the system is a shared-bus cache-coherent multiprocessor (SMP) that maintains coherence among caches with a snooping protocol. Each node consists of c aggressive 400 MHz, 1 CPI processors, with an 8 cache-line deep merging write-buffer and with a two-level cache hierarchy; a 16KB direct-mapped first-level data cache and a 1MB two-way set-associative second-level cache (with a 64 byte line size and a 10-cycle hit cost). The cache policy can be programmed on a per-page basis to be either write-back with write-allocate or write-through with write-around. The latter is used only for shared pages, so automatic updates may be detected. Each node has an interleaved memory with a 200ns latency. The write buffers merge writes and retire them in FIFO order. All instructions in our simulated processors as well as cache hits cost 1 CPU cycle. We use a 400MHz processor to capture the event of out of order execution in the last generation processors, that increases the load on the memory bus and the memory subsystem. The TLB invalidations within the cluster discussed earlier are not modeled yet in the simulator. In this sense, the simulations are somewhat optimistic for MP-AURC, since including the effects and the costs of TLB invalidations will hurt its performance relative to AURC.

The shared bus within each node is a split transaction bus. All devices release the bus after posting a request; additionally, devices (e.g. DRAM) that receive multiple requests use a FIFO queue to store and service them in order. The priority of each device in the node for bus arbitration is as follows; the incoming network path has highest priority, then the write buffer, the cache, the DRAM and finally the outgoing network path. This order was chosen since it delivered the best MP-AURC performance in several tests.

The nodes (SMPs) are connected with a high speed interconnect via an automatic-update-capable network interface plugged into the I/O bus of the node. Node-to-network bandwidth is therefore limited by the speed of the I/O bus. The overhead of initiating a deliberate transfer (e.g. a page transfer) is 1 microsecond, and the overhead of initiating an automatic update is a single CPU cycle when the write buffer is not full. The page size is 4KB, so a page transfer takes 1K I/O bus cycles. The network interface has a mechanism that combines consecutive automatic updates to a single network packet, whose size cannot exceed 40 words. The network topology is a two-dimensional mesh. The simulator models contention everywhere (buses, memory, network interface) in great detail, except for contention in the network topology itself.

We model a fast interrupt handling mechanism which takes 150 cycles. All protocol operations are actually simulated by running the protocol handlers, and the simulator also charges for the cache misses incurred during the operations.

We set the number of processors per cluster to 4 and the total number of processors to 16. We plan to address this part of the configuration space in future work.

The base architecture has an 800MB/s memory bus and a 256MB/s I/O bus. We use the same values for the uniprocessor and multiprocessor configurations because it likely that the same components will be used in building one to few processor extendible commodity systems.

Finally, to obtain an estimate of just how much node-to-network (here I/O bus) bandwidth is enough we look at a much more aggressive interconnect system with a memory bus of 1.6GB/s and an I/O bus of 1.6GB/s. Since the network interface is connected to the I/O bus, the bandwidth of the network interface is limited by the I/O bus bandwidth. Increasing the network interface bandwidth can be done either by increasing the I/O bus bandwidth or by adding more I/O buses and network interfaces to the node. Since adding multiple network interfaces creates ordering problems we assume that the available network interface bandwidth is increased by using faster I/O buses. While this I/O bus system is not commodity today, it helps identify problems associated with building an SVM system with SMP nodes.

5 Applications

To study the architectural, protocol and application implications of building shared memory parallel systems out of commodity SMPs, we conducted a series of simulations of the MP-AURC and AURC protocols for the basic architecture model we described in Section 3. We chose to run a subset of Splash-2 [13] applications and kernels with various communication to computation ratios and sharing patterns. Table 1 shows the speedups achieved for these applications under AURC.

LU is a kernel which factors a dense matrix into the product of a lower triangular and an upper triangular matrix. In our simulations we used a 512x512 matrix. The application exhibits a very small communication to computation ratio but it is inherently imbalanced. The speedup for 16 uniprocessor nodes running AURC is about 9.

Water-nsquared is a molecular dynamics simulation, similar to the original Water code from the Splash-2 suite. We use a data set size of 343 molecules. Updates are accumulated locally between iterations, and performed at once at the end of each iteration which results in a small communication to computation ratio. The speedup for 16 uniprocessors node running AURC is about 9.

Ocean is a fluid dynamics application which simulates large scale ocean movements. We simulate a square grid of size 258 by 258 points with the error tolerance for iterative relaxation set to 0.001. Ocean is a representative of the class of problems based on regular-grid computation. At each iteration the computation performed on each element of the grid requires the values of its four neighbors. The speedup for 16 uniprocessor nodes running AURC is about 6, but it grows quickly with larger problem sizes [7].

FFT is a high-performance FFT kernel in which the communication is essentially a matrix transposition of a 512 by 512 matrix of complex numbers(256K FFT). The write access granularity is coarse while the read access granularity is relative smaller which produces a relative large communication to computation ratio for this problem size. The speedup for 16 uniprocessor nodes running AURC is about 5, but again grows quickly with larger problems [7].

Barnes is an irregular application which simulates the interactions among a system of particles over a number of time steps, using the Barnes-Hut hierarchical N-body method. We used a data set size of 8K particles. Both read and write access patterns to the global particle and cell arrays are fine-grained which causes substantial fragmentation and false sharing to occur under AURC. The communication to computation ratio is medium with a speedup of about 9 for 16 uniprocessor nodes under AURC.

6 Protocol Performance

We compare the performance of MP-AURC and MP-LRC on a 4 cluster configuration with 16 total processors (4x4) against the performance of AURC and LRC protocols running on 16 uniprocessor nodes. For AURC and MP-AURC we consider versions with(AURC, MP-AURC) and without(AURC-2, MP-AURC-2) the Copyset-2 scheme. As mentioned above for the extended multiprocessor protocols (MP-LRC, MP-AURC-2) we also investigate a configuration with a memory and I/O bandwidth of 1.6GB/s.

Tables 1 and 2 present the speedups and number of page faults for the five applications. The first three columns contain data for the uniprocessor node protocols(LRC, AURC, AURC-2), the next three refer to the extended protocols (MP-LRC, MP-AURC, MP-AURC-2) and the last two columns (F-MP-LRC, F-MP-AURC-2) report the speedups for MP-LRC and MP-AURC in the configuration with the increased bandwidth.

Previous work [7] has shown that AURC outperforms LRC (columns 1 and 2) for the uniprocessor nodes. For the applications considered here, the performance gap is substantial only for Barnes and Ocean. For the architectural parameters chosen for these simulations, the effect of Copyset-2 scheme is contradictory. The only applications for which the Copyset-2 scheme is effective in saving page faults is Ocean (Table 2). However for all applications (including Ocean), the Copyset-2 scheme has small or insignificant contribution to the overall performance.

The relative performance of the extended protocols (MP-AURC and MP-LRC) compared to uniprocessor node protocols (AURC and LRC), is mainly dictated by the communication to computation ratio across nodes (clusters) and by how bursty the communication is. For LU and Water-Nsquared, that are applications with small communication to computation ratio, MP-LRC and MP-AURC perform very similarly to LRC and AURC respectively. Node-to-network bandwidth requirements do increase with clustering, but they are still low enough that the available bandwidth suffices. The imbalance in the LU computation is still present across the clusters. Increasing the bandwidth of the I/O bus (hence the node-to-network interface) brings the extended protocols to slightly better performance compared to the uniprocessor node case.

The other two applications (Ocean and FFT) with relatively larger communication to computation ratios are more interesting. They also exhibit communication patterns that are commonly found in scientific computing. Although we expect a decrease in the number of page faults due to intra-cluster coherence (Table 2), we also expect an increase of the bandwidth requirements when processors are clustered in SMPs, as mentioned before. Computation time requirements per cluster increase faster (linearly) than per processor cross-node traffic decreases with increasing cluster size. Thus the bandwidth requirements per network interface increase. Both applications benefit from clustering under LRC but not under AURC. For Ocean, the LRC speedup increases from 4.39 to 5.82 when extended to SMP while the speedup of AURC decreases from 10.79 to 6.06 (still better than both LRC protocols). Similarly for FFT, the LRC speedup increases from 5.61

Application	LRC	AURC	AURC-2	MP-LRC	MP-AURC	MP-AURC-2	F-MP-LRC	F-MP-AURC-2
LU	8.66	10.23	10.44	8.88	10.40	10.44	9.19	10.54
Water	7.32	9.25	9.24	7.12	9.10	9.08	7.43	10.29
Barnes	3.09	10.03	10.01		9.63	9.63		11.58
Ocean	4.39	10.79	11.34	5.82	6.06	9.93	7.62	12.24
FFT	5.61	7.32	7.24	6.33	2.38	2.76	13.81	4.05

Table 1: Application Speedups for the different protocols.

Application	LRC	AURC	AURC-2	MP-LRC	MP-AURC	MP-AURC-2	F-MP-LRC	F-MP-AURC-2
LU	99	99	99	99	99	99	99	99
Water	191	160	159	173	117	117	184	123
Barnes	1127	864	867		738	759		721
Ocean	1499	1251	1580	387	234	1228	335	1316
FFT	1440	1440	1440	1272	1216	1344	1272	1344

Table 2: Average Number of Page Faults per processor for the different protocols.

to 6.33 by using SMP clusters while the AURC performance goes down dramatically from 7.32 to 2.38. Running AURC without the Copyset-2 scheme tends to perform better; for Ocean the speedups recover from 6.06 to 9.93, while for FFT it increases from 2.38 to 2.76. As expected, the clustering effect amplifies the use of Copyset-2 for up to 8 processors as long as they are contained in no more than 2 nodes. This can result in more savings in the number page faults (a factor of 5 in Ocean , Table 2). However, the extra communication contention due to automatic update traffic in Copyset-2 is more harmful for the overall performance in MP-AURC than in AURC. Finally, a higher memory and I/O bus bandwidths turns to be much more useful for FFT than for Ocean and for MP-LRC more than for MP-AURC.

In the following section we present in detail the simulation results for Ocean and FFT, two applications that are particularly interesting because of their communication needs. Similar graphs are available for all the applications in tables 1 and 2 but we do not present them due to space limitations.

6.1 Performance Analysis

As mentioned above, Ocean and FFT are two applications with relatively high communication demands. To understand their behavior under each protocol we present two types of graphs. The first type shows the breakdown of the execution time for each of the 16 processors in each case. The components of the execution time are: the compute time of each processor (Thread Compute Time), the time spent waiting for page fetches after a page fault (Thread Data Wait Time), the time spent in locks (Thread Lock-WN Wait Time), the time spent in barriers (Thread Barrier-WN Wait Time), the time spent in protocol handlers or protocol overhead (Handler Compute Time) and the time each processor is stalled in the local memory subsystem, including caches, write buffers, memory bus arbitration and the local memory itself (Total CPU-Cache Stall Time). The units used are simulation cycles. By comparing execution times on the Y-axis across graphs, we can tell whether AURC or MP-AURC performs better [3]. The second type presents the amount of data communicated in various parts of the system. This includes the amount of data received by each processor from other nodes, the amount of data sent with deliberate update, the traffic between the local memory and the caches and the write buffer to local memory traffic. The first two of these represent the load on the I/O bus, whereas the other two is the load on the memory bus. The data received by each node are attributed to the requesting processor.

First we give a more analytical substantiation of the performance behavior for Ocean and FFT. For nearest-neighbor communication (Ocean) consider a k node system with c processors per node, thus a total of $p = k * c$ and an n-by-n grid. With $c = 1$ processor per cluster, the computation time is $O(n^2/p)$ (a processor's partition area) and the cross-cluster communication volume is $O(n/\sqrt{k}) = O(n/\sqrt{p})$ (the perimeter of the total partition assigned to all processors in the cluster, here 1). With c processors per cluster, computation time remains the same, while cross-cluster communication volume is now $O(n/\sqrt{k}) = O(n/\sqrt{p/c})$

since there are $k = p/c$ clusters. Thus, even ignoring intra-cluster communication and its interference with the cross-cluster communication, the inter-node communication bandwidth needs are greater by a factor of $\sqrt{c}$.

For all-to-all communication (FFT) let $D = n/p$ be the number of points per processor. With $p = 1$ processor per cluster, computation time is $O(D\ log\ D)$ and SVM communication volume is $O(\frac{D}{p} * (p-1)) = O(\frac{n*(p-1)}{p^2})$. With c processors per cluster, computation time is the same. Cross-cluster communication volume from the p processors in the node is $O(\frac{c*D}{k} * (k-1)) = O(\frac{c*\frac{n}{p}}{\frac{p}{c}} * (\frac{p}{c} - 1)) = O(c * \frac{n*(p-c)}{p^2})$, which is substantially larger (by almost a factor of c) than in the uniprocessor node case.

Figures 5–9 show the time breakdown in the different protocols for Ocean. The total execution time under MP-AURC increased almost by a factor of 2 compared with the execution time under AURC, although the number of page faults went down by a factor of 5 due to clustering. The main reason for this increase in the execution time is the increased automatic update traffic. This is because of the Copyset-2 protocol. Since shared pages are mapped bidirectionally all the writes to shared pages from every processor in the two nodes that participate in the mapping are propagated to the remote node. This means that the automatic update traffic includes not only the boundary elements in the nearest neighbor communication but all the elements in shared pages that contain boundary elements. Since there are four processors per node using the Copyset-2 scheme imposes a significant load to the memory and I/O buses. Figures 10–14 show the traffic on the memory and the I/O bus for each protocol. We see that the network traffic in the AURC protocols is caused almost exclusively by automatic update. This is the reason, that MP-AURC-2 performs so much better than MP-AURC, and much closer to AURC and AURC-2. In this case, given the limited memory and I/O bandwidths the Copyset-2 extensions has a clearly negative effect. If we disable Copyset-2 then MP-AURC-2 performs much better than MP-LRC but still worse than AURC. Increasing the memory and I/O bandwidth improves the performance of both MP-AURC-2 and MP-LRC with the performance improvement being bigger in MP-LRC.

Figures 15–19 show the time breakdown for FFT in the different protocols. The execution time under MP-AURC increases by a factor of 3 when the I/O bus bandwidth is kept the same. The communication cost is on average 2-3 times greater than in AURC and non-uniformly distributed within the cluster. The number of page faults went down by 14% (since here there are few clusters, the drop is large) but unlike in ocean the distribution of page faults across processors is perfectly uniform. This means that the imbalance is caused exclusively from bus and network interface contention rather than from the protocol or the application. Clustering reduces the traffic substantially in both AURC and LRC (Figures 20–24). This is mainly because of prefetching effects. Since the communication pattern is all-to-all when a processor in a node fetches the data it needs it prefetches also data for other processors in the same node. The exact amount of prefetching depends on the problem size and the alignment of data. Eliminating the Copyset-2 scheme doesn't help much because most of the traffic is due to page fetches (deliberate update). Each processor writes pages that are owned by the local node and reads data from every other processor. Thus, both the automatic update traffic and the applicability of the Copyset-2 scheme are low. A 1.6 GB/sec I/O (and memory) bus improves performance but the execution time is still a factor of 2 bigger than AURC. We observe that the data wait time decreases but not as much as expected given the extremely high bandwidth of the I/O Bus. We also note that there is a big imbalance in the page wait times between nodes (Figure 16). This imbalance is caused by the contention in the memory bus. After the processors in a node receive the page they need they start computing which generates a lot of memory bus traffic because of the write-through shared memory pages. Since the cache and the write buffer in a node have higher priority over the outgoing path of the network interface for the memory bus, this node will respond to page requests with delay. Thus the requesting nodes incur higher page wait delays. This imbalance is then carried to synchronization time since some of the nodes finish processing earlier than others they wait at barriers until everybody gets to that point. In contrast to Ocean, in FFT the traffic on the I/O bus (and thus the contention in the network interface) is caused by both automatic and deliberate update traffic. Since there is a fixed overhead to initiate a deliberate/automatic update transfer and the requests are serviced in FIFO order, page requests may incur high delays. This is the reason that the data wait time is not decreased as much as expected. The long delays in page requests cause imbalances among nodes and consequently the synchronization time is also increased significantly.

In FFT's all-to-all communication, contention is also caused at the node-to-network interface either because algorithmically several processors within a cluster need to communicate with processors in another cluster at the same time, or the processors that currently communicate only within the cluster get ahead of those that communicate off-cluster at a give stage of the algorithm, and so all start to communicate off-cluster at the same time. The exact pattern of which processors have to wait due to contention is timingdependent. The contention may be alleviated somewhat by changing the order in which processors communicate with each other during the all-to-all communication phase to take advantage of the two-level communication hierarchy. Increasing the problem size does not change the communication to computation ratio quickly in FFT, and at larger problems capacity misses on the bus interfere more with communication, so larger problems are not likely to help MP-AURC relative to AURC much if at all.

Although we don't have yet a full evaluation to report, we believe that some irregular applications may be better candidates for MP-AURC, mainly because their fine-grained sharing can benefit from the efficient and fine-grained intra-cluster coherence. This same communication at the SVM page-grain is expensive, so the savings due to clustering can be large. Preliminary results from running the Barnes-Hut under MP-AURC (Table 1) show that MP-AURC performs very close to AURC. There is a slight increase in synchronization cost (due to increased imbalance induced by the variability in communication costs) but communication is reduced considerably relative to the base AURC. We are in the process of evaluating more applications, particularly irregular ones, and will be able to report on the results in the final version of the paper, if accepted.

7 Related Work

Cox et al. in [2] discuss three different shared memory configurations. For small numbers of processors they compare a bus based cache coherent shared memory system to a SVM implementation (TradeMarks). For larger numbers of processors they simulate an all-hardware (AH) cache coherent shared memory system, an all-software (AS) SVM system (TradeMarks) and a hardware-software (HS) SVM system running TradeMarks on top of SMPs with 8 processors per node. In this work no hardware support is assumed for the SVM layer, whereas we consider an architecture that uses automatic update support.

They find that for applications with good locality (small communication needs) and moderate synchronization, HS results in performance comparable to AH. Also for small numbers of processors and applications with moderate communication and synchronization needs AS performs comparable to AH. They also report that HS and AS do not scale well, unless the software overheads (communication, interaction with the operating system etc.) are reduced.

Karlsson et al. in [8] evaluates the performance of a SVM multiprocessor configuration built exclusively with commodity parts. They consider a cluster-based multiprocessor built from ATM switches and bus-based SMPs. As in [2], the protocols used are essentially the ones in TreadMarks. They find that, although the bandwidth of present as well as next generation ATM technology is enough, the major bottleneck lies in current ATM interfaces whose latencies are not acceptable. By tailoring the network protocols for the SVM system, they drastically reduce the overheads. They also investigate the SVM protocol overheads and find out that it is not necessary to dedicate a processor for protocol handling.

8 Future Work

We are currently working on more detailed simulation models that will allow us to investigate the configuration space for SMP-based multiprocessors. Directions for future work on the architectural aspects include system configurations with larger numbers of processors and varying numbers of processors and network interfaces (I/O buses) per node. Also, network interface configurations with multiple queues and different priorities for different types of packets. Finally on the protocol and application sides, more work is needed to thoroughly understand the interactions of more classes of applications with the systems and how to restructure applications to take advantage of the two-level hierarchy.

9 Conclusions

We have shown that constructing large-scale shared address space multiprocessors by using a looser SVM layer to connect hardware-coherent commodity SMPs can be done using AURC. This requires some extensions to the protocol to support SMP rather than uniprocessor nodes and some revisions of lower-level decisions(e.g. protocol data structures). Although AURC performs better than LRC for the uniprocessor case, the extended protocol, MP-AURC, while still better than MP-LRC in most cases, performs comparably to AURC only for applications that don't communicate much. For many important applications MP-AURC requires higher bandwidth per node than MP-LRC due to the write-through and AU traffic. This can be partially dealt with by eliminating the Copyset-2 scheme from AURC and using higher I/O and memory bus bandwidths. If the network interface bandwidth is the same for both uniprocessor and multiprocessor nodes, AURC is the protocol that performs best for the set of applications we investigate. Memory and I/O bus contention in MP-AURC cause load imbalances in certain applications which further reduces performance.

Despite these technical problems, the marketplace advantages are significant. MP-AURC works very well for applications that don't communicate much, and is likely to work better for large problem sizes despite greater interference on the bus from capacity misses. Restructuring applications to take advantage of the two-level communication hierarchy is also possible in some cases, and we have not done it here. Nonetheless, our results suggest that the straightforward extension of the systems is not sufficient for existing applications, and much further work is needed on the systems, evaluation and applications fronts. In fact, shared address space multiprocessors that easily provide more network interfaces from the node (such as DSM machines [14]) may provide more likely candidates for the building blocks.

References

[1] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[2] 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 Symposium on Computer Architecture*, pages 106–117, April 1994.

[3] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 144–155, May 1993.

[4] A. Erlichson, B. Nayfeh, J.P Singh, and K. Olukotun. The Benefits of Clustering in Shared Address SPace Multiprocessors: An Applications-Driven Investigation. In *Supercomputing '95*, pages 176–186, 1995.

[5] S.A. Herrod. *TangoLite: A Multiprocessor Simulation Environment.* Computer Systems Laboratory, Stanford University, 1994.

[6] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[7] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[8] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[9] 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 USENIX Conference*, pages 115–132, January 1994.

[10] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[11] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve t he Performance of Distributed Shared Memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[12] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 325–336, April 1994.

[13] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.

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

[15] M.J. Zekauskas, W.A. Sawdon, , and B.N.Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 87–100, November 1994.

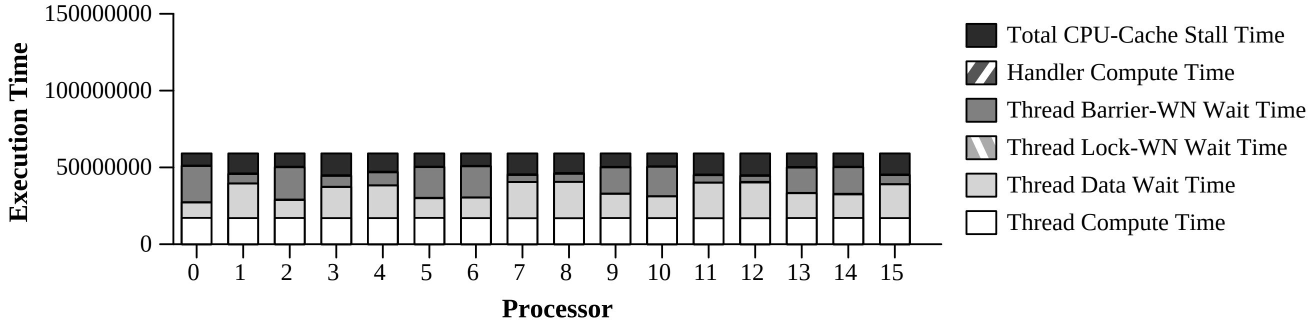

Figure 5: Execution Time Breakdown for Ocean, AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

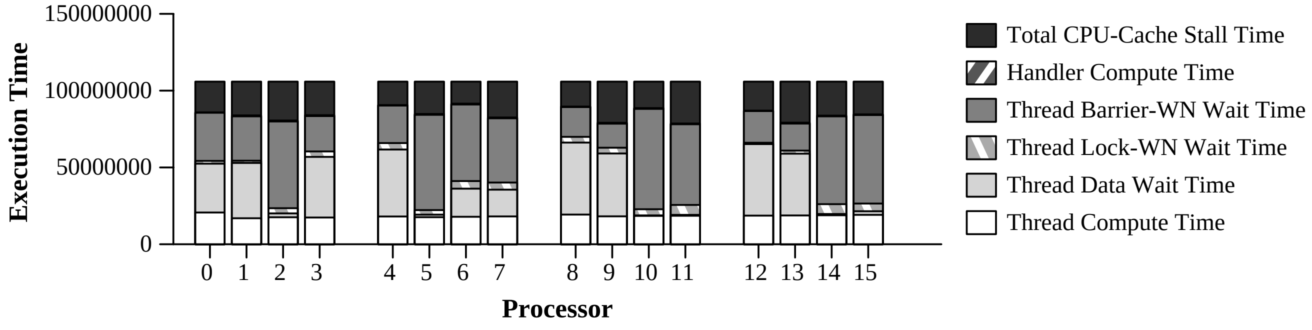

Figure 6: Execution Time Breakdown for Ocean, MP-AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

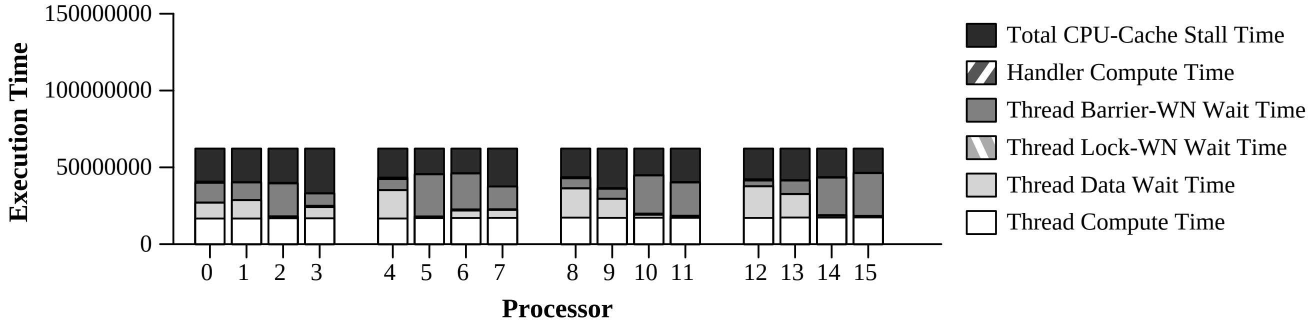

Figure 7: Execution Time Breakdown for Ocean, MP-AURC-2, 800MB/s Memory Bus, 256MB/s I/O Bus.

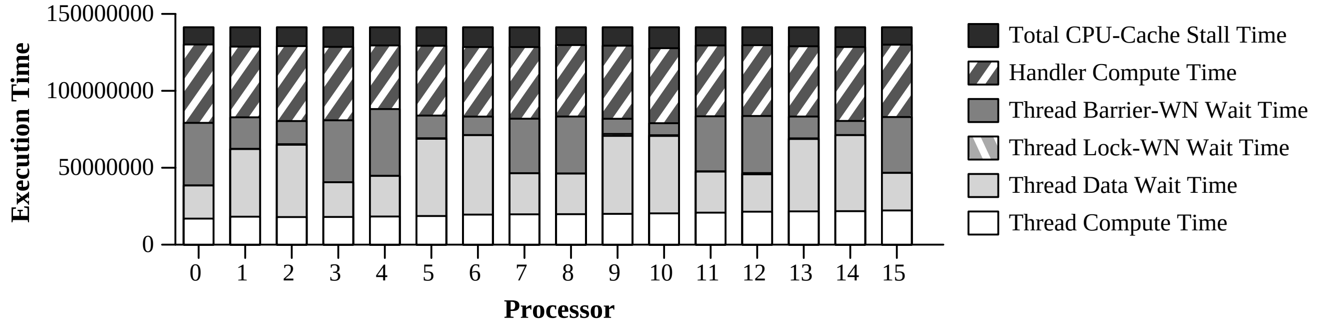

Figure 8: Execution Time Breakdown for Ocean, LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

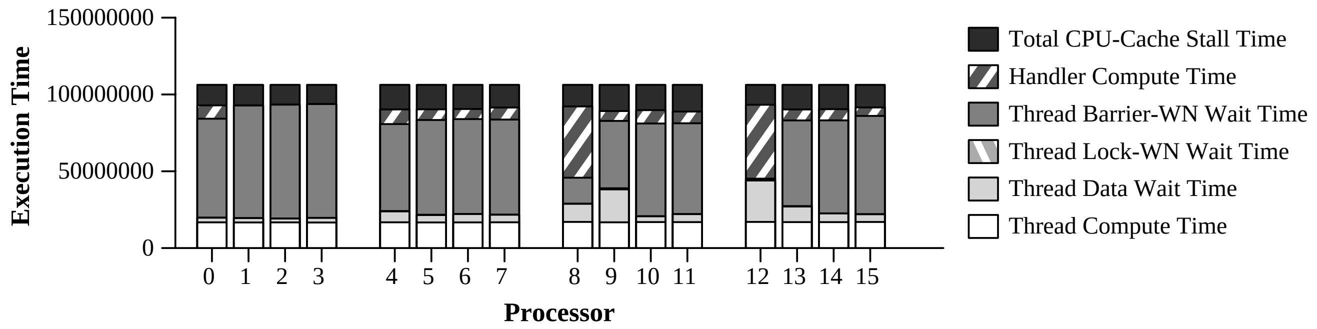

Figure 9: Execution Time Breakdown for Ocean, MP-LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

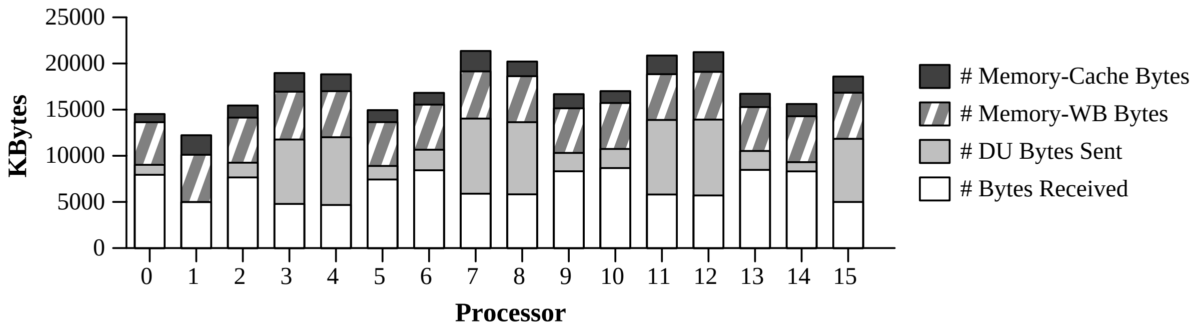

Figure 10: Traffic in Bytes for Ocean, AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

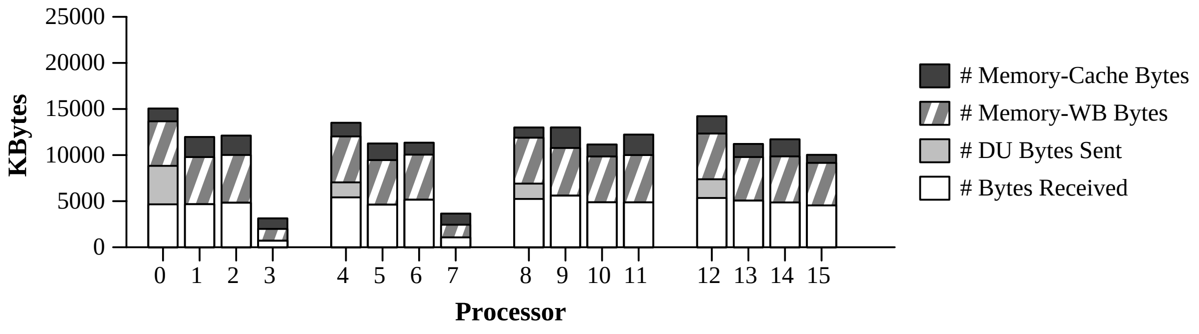

Figure 11: Traffic in Bytes for Ocean, MP-AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

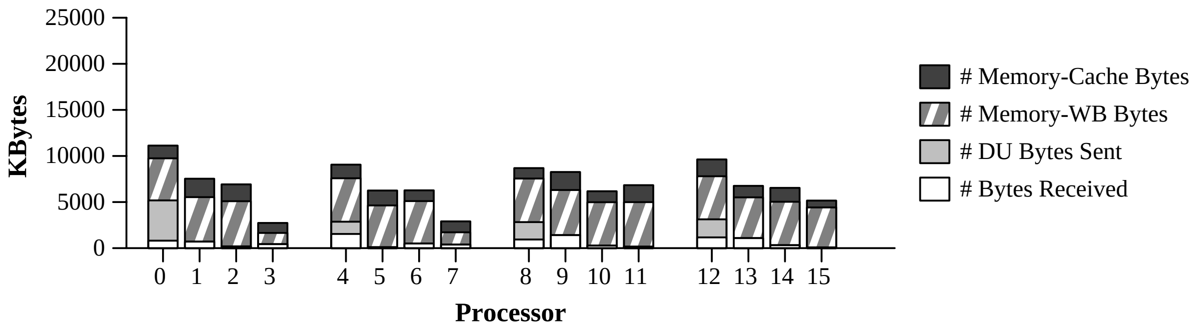

Figure 12: Traffic in Bytes for Ocean, MP-AURC-2, 800MB/s Memory Bus, 256MB/s I/O Bus.

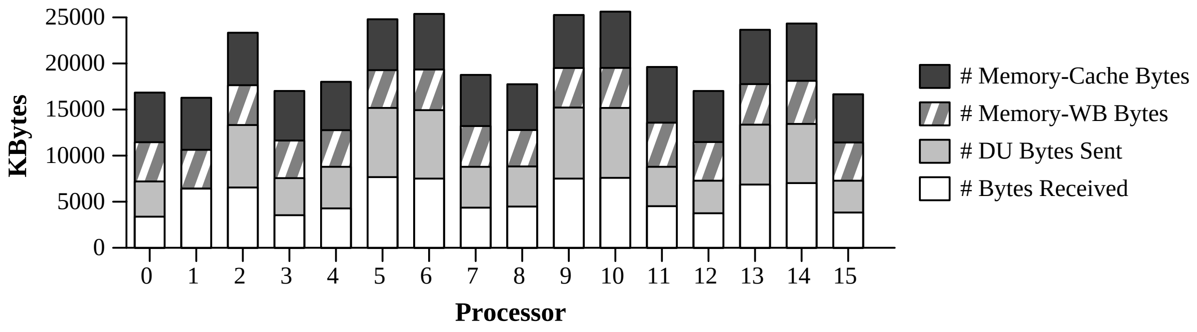

Figure 13: Traffic in Bytes for Ocean, LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

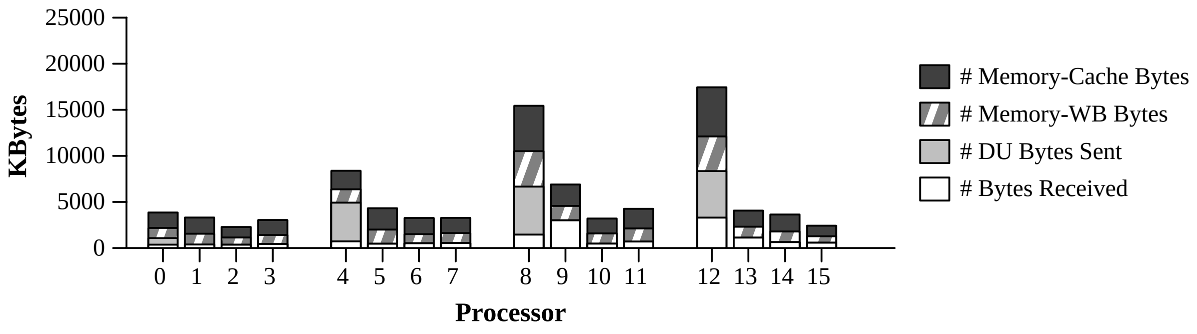

Figure 14: Traffic in Bytes for Ocean, MP-LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

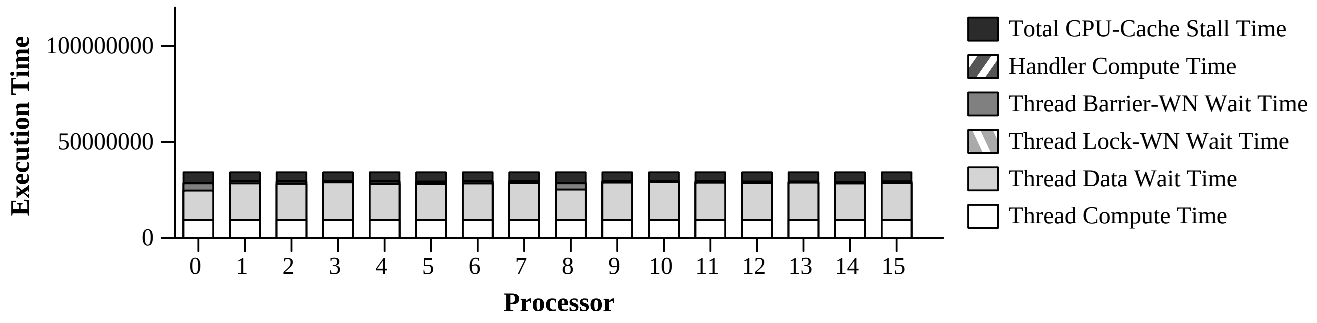

Figure 15: Execution Time Breakdown for FFT, AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

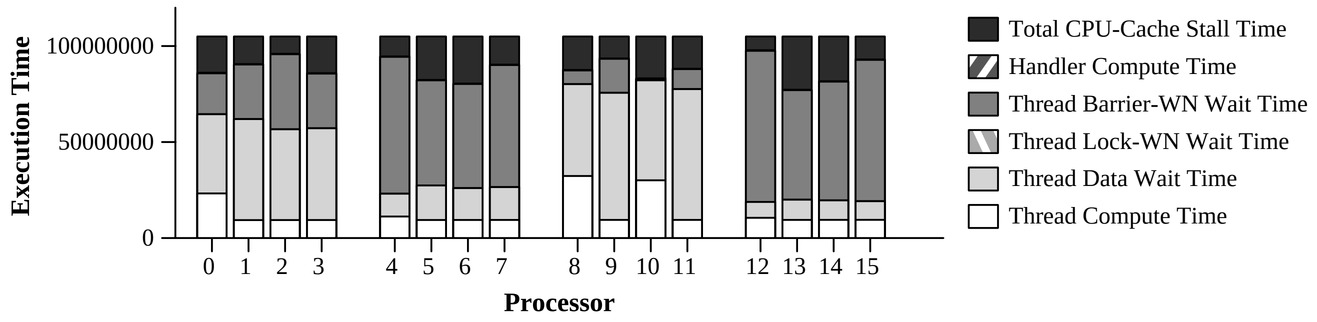

Figure 16: Execution Time Breakdown for FFT, MP-AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

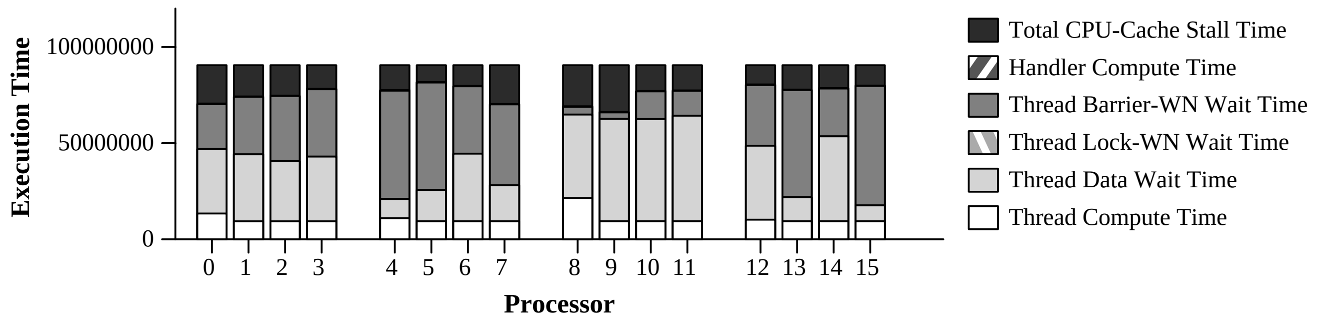

Figure 17: Execution Time Breakdown for FFT, MP-AURC-2, 800MB/s Memory Bus, 256MB/s I/O Bus.

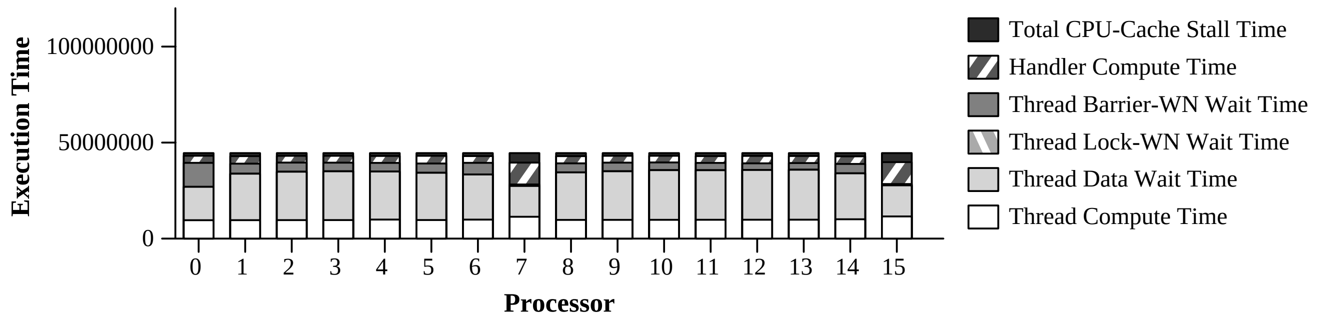

Figure 18: Execution Time Breakdown for FFT, LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

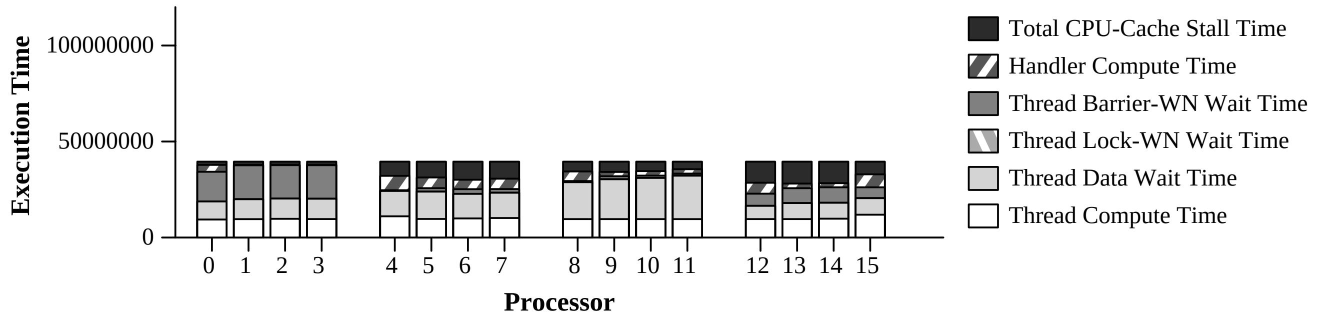

Figure 19: Execution Time Breakdown for FFT, MP-LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

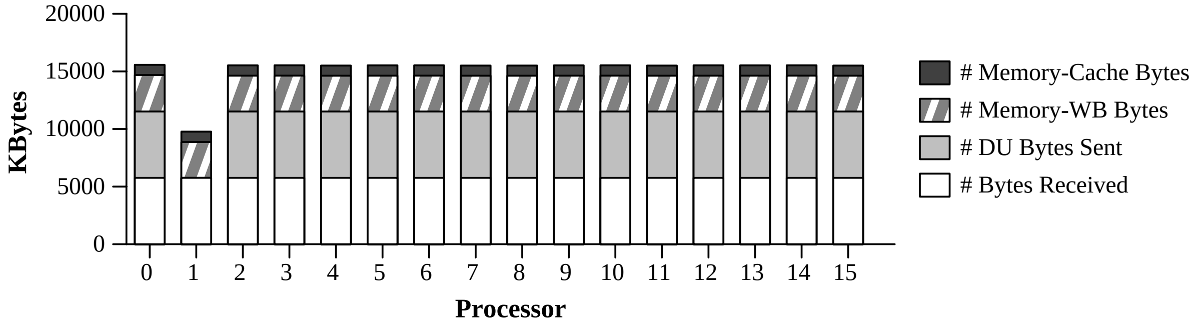

Figure 20: Traffic in Bytes for FFT, AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

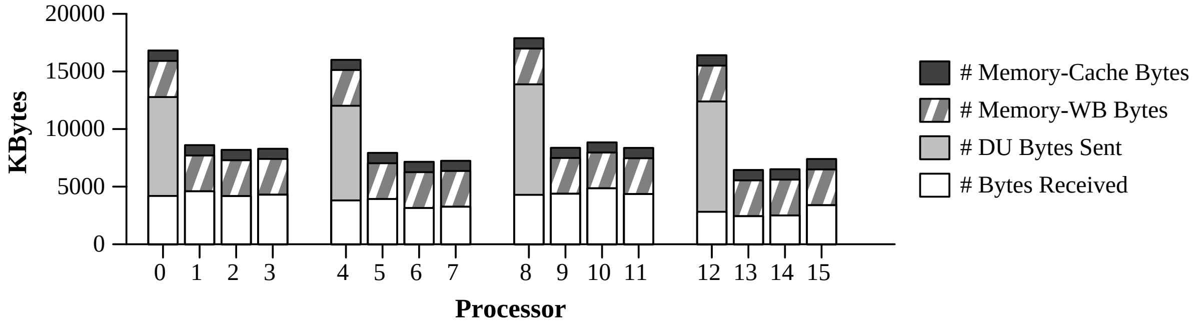

Figure 21: Traffic in Bytes for FFT, MP-AURC, 800MB/s Memory Bus, 256MB/s I/O Bus.

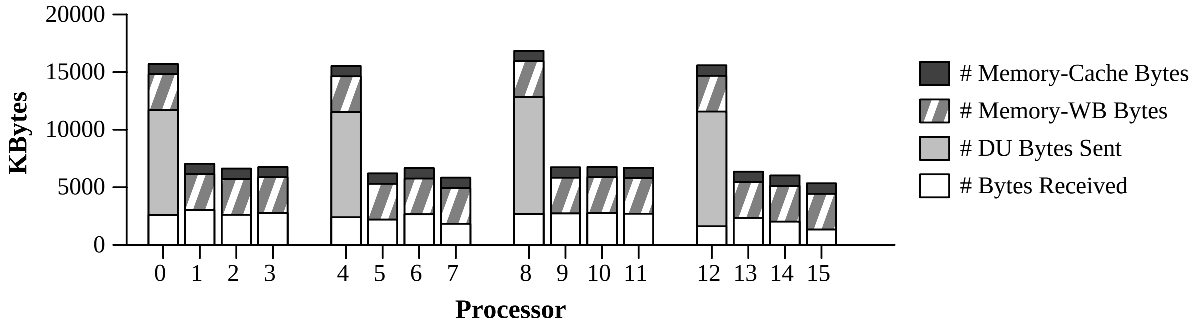

Figure 22: Traffic in Bytes for FFT, MP-AURC-2, 800MB/s Memory Bus, 256MB/s I/O Bus.

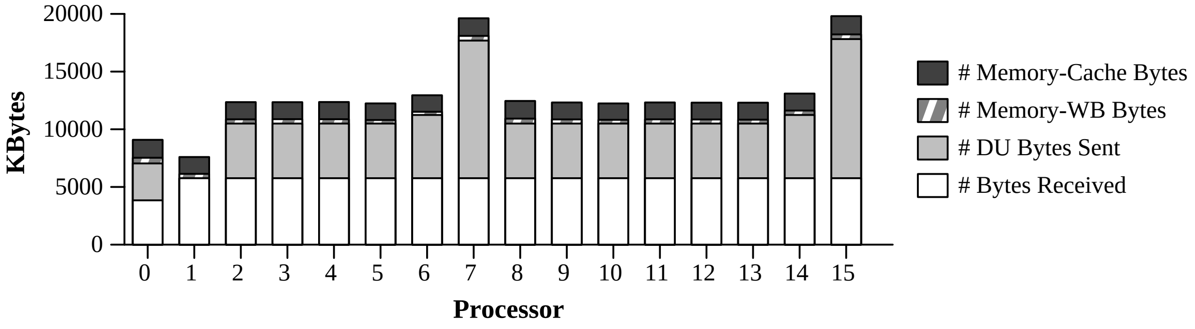

Figure 23: Traffic in Bytes for FFT, LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.

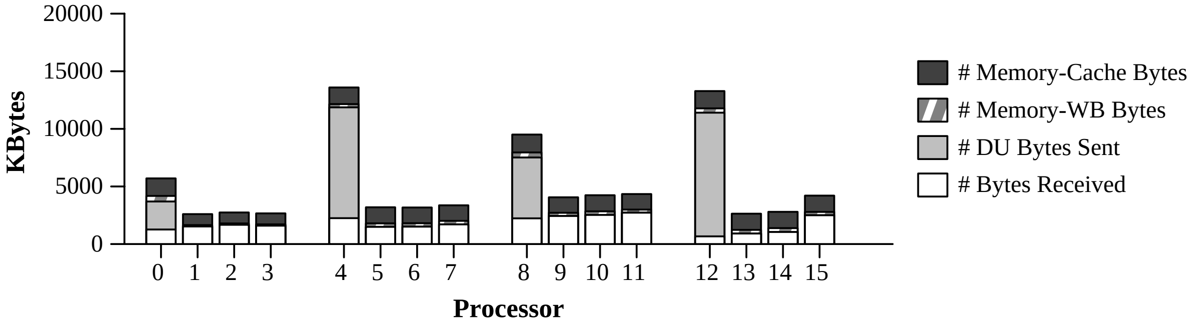

Figure 24: Traffic in Bytes for FFT, MP-LRC, 800MB/s Memory Bus, 256MB/s I/O Bus.