COMIC++: A Software SVM System for Heterogeneous Multicore Accelerator Clusters*

Jaejin Lee† Jun Lee† Sangmin Seo† Jungwon Kim† Seungkyun Kim† Zehra Sura‡

†School of Computer Science and Engineering, Seoul National University, Seoul, Korea
‡IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA
jlee@cse.snu.ac.kr, {jun, sangmin, jungwon, seungkyun}@aces.snu.ac.kr, zsura@us.ibm.com
http://aces.snu.ac.kr

Abstract

In this paper, we propose a software shared virtual memory (SVM) system for heterogeneous multicore accelerator clusters with explicitly managed memory hierarchies. The target cluster consists of a single manager node and many compute nodes. The manager node contains a general-purpose processor and larger main memory, and each compute node contains a heterogeneous multicore processor and smaller main memory. These nodes are connected with an interconnection network, such as Gigabit Ethernet. The heterogeneous multicore processor in each compute node consists of a general-purpose processor element (GPE) and multiple accelerator processor elements (APEs). The GPE runs an OS and the multiple APEs are dedicated to compute-intensive workloads. The GPE is typically backed by a deep on-chip cache hierarchy and hardware cache coherence. On the other hand, the APEs have small explicitly-addressed local memory instead of caches. This APE local memory is not coherent with the main memory. Different main and local memory units in the accelerator cluster can be viewed as an explicitly managed memory hierarchy: global memory, node local memory, and APE local memory. Since coherence protocols of previous software SVM proposals cannot effectively handle such a memory hierarchy, we propose a new coherence and consistency protocol, called hierarchical centralized release consistency (HCRC). Our software SVM system is built on top of HCRC and software-managed caches. We evaluate the effectiveness and analyze the performance of our software SVM system on a 32-node heterogeneous multicore cluster (a total of 192 APEs).

1. Introduction

For some application domains, multicore accelerator architectures such as GPUs and Cell BE processors have emerged to speedup computationally intensive parts of applications. These architectures are widening their user base to all computing domains, resulting in a new trend in high performance computing systems where a cluster system now consists of heterogeneous nodes that are built from multicore processors and accelerators. A good example is the IBM Roadrunner cluster at the Los Alamos National Laboratory[20].

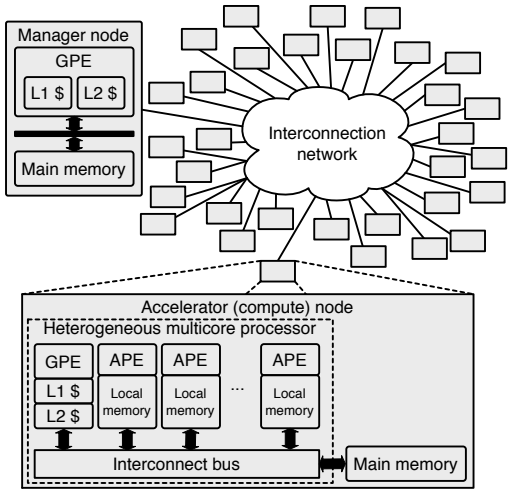


Figure 1. The target accelerator cluster architecture.

Multicore accelerator systems typically have explicitly managed memory hierarchies, where the programmer or software is responsible for explicitly moving data between different levels of the memory hierarchy. Traditionally, hardware caches and their coherence mechanism provide the programmer or software with the abstraction of a single shared address space, resulting in ease of programming. With an explicitly managed memory hierarchy, the burden of managing coherence between different memory locations shifts to the programmer. However, explicit memory management gives the programmer or software more control in managing the costs of data movement, and a greater scope to improve performance through sophisticated optimizations. Therefore, an important problem for contemporary multicore accelerator cluster architectures is to develop a shared memory interface that is efficient for explicitly managed memory hierarchies, scalable to a large number of cores and nodes in the system, and easy to program for software development processes.

In this paper, we introduce a software shared virtual

*This work was supported in part by the Creative Research Initiatives (Center for Manycore Programming, 2009-0081569) of MEST/KOSEF and by the Ministry of Education, Science and Technology under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

987-1-4244-5659-8/09/$25.00 ©2009 IEEE

memory (software SVM) system, called COMIC++, for heterogeneous multicore accelerator clusters, and describe its associated memory consistency model. COMIC++ provides the programmer with an illusion of a single globally shared address space by hiding the distributed nature of the underlying memory system.

The target cluster architecture is shown in Figure 1. It is similar to the organization of asymmetric chip multiprocessors (ACMPs)[14, 21, 24], which consists of one large, high performance core that boosts the execution of serial program portions and many small cores that execute parallel portions to achieve high performance. The target cluster consists of one manager node and many compute (accelerator) nodes. The nodes are connected by an interconnection network, such as Gigabit Ethernet.

The manager node has a high performance general-purpose processor and each compute node has a heterogeneous multicore processor. The heterogeneous multicore processor consists of a general-purpose processor element (GPE) that runs an OS and multiple accelerator processor elements (APE) dedicated to compute-intensive workloads. The PEs are connected by a high-speed interconnect bus. A typical example of the accelerator node architecture is the Cell BE architecture[13, 17]. Another example is an architecture in which a general-purpose processor is connected to multiple accelerator cards. The accelerator card typically has GPGPUs[26, 31] or homogeneous manycore processors, such as Tilera TILEPro64[34].

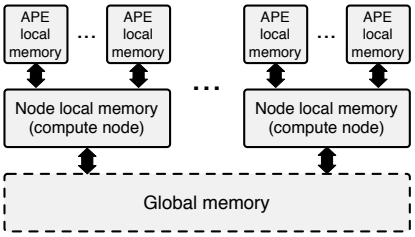


Figure 2. The explicitly managed memory hierarchy.

Different memories in the cluster architecture can be viewed as an explicitly managed memory hierarchy as shown in Figure 2: *global memory*, *node local memory*, and *APE local memory*. Coherence and consistency between different memories in different levels in the hierarchy are not supported by hardware. Instead, COMIC++ guarantees coherence and consistency. It does this efficiently by using a protocol that distinguishes between intra-node and inter-node events for coherence or consistency, and by adapting the memory consistency implementation to the target architecture (either the intra-node or inter-node architecture).

Our approach significantly differs from previous software SVM proposals [2, 6, 18, 19, 23, 29, 30, 33, 36, 37] for loosely coupled, distributed memory multiprocessors, such as clusters of workstations or PCs. First, COMIC++ handles coherence and consistency both at the same level and across different levels for the explicitly managed memory hierarchy, while previous proposals target explicitly managed memories at the same level only. Second, unlike previous proposals, our protocol does not require complicated management of synchronization intervals and vector timestamps to obtain happens-before ordering between events. Third, while previous proposals exploit an MMU in each cluster node to detect and handle page faults, COMIC++ does not exploit any hardware MMU. This is because at the heterogeneous multicore level, typically there is no page-fault handling MMU functionality implemented for APEs. Fourth, COMIC++ guarantees page-level coherence and consistency with any page size while the page size of previous SVM proposals is restricted to the page size of the OS. Fifth, unlike previous proposals in which there is almost no limitation to the size of available memory due to virtual memory provided by the OS in each node, COMIC++ targets a memory hierarchy in which neither the hardware nor the OS supports virtual memory at some level in the hierarchy. COMIC++ uses software-managed caches[9, 12, 32] implemented in each level of the memory hierarchy. Finally, polling or interrupt handling to process incoming, asynchronous messages in the APE is quite expensive. In COMIC++, only an APE initiates communication for coherence and consistency.

The major contributions of this paper are the following:

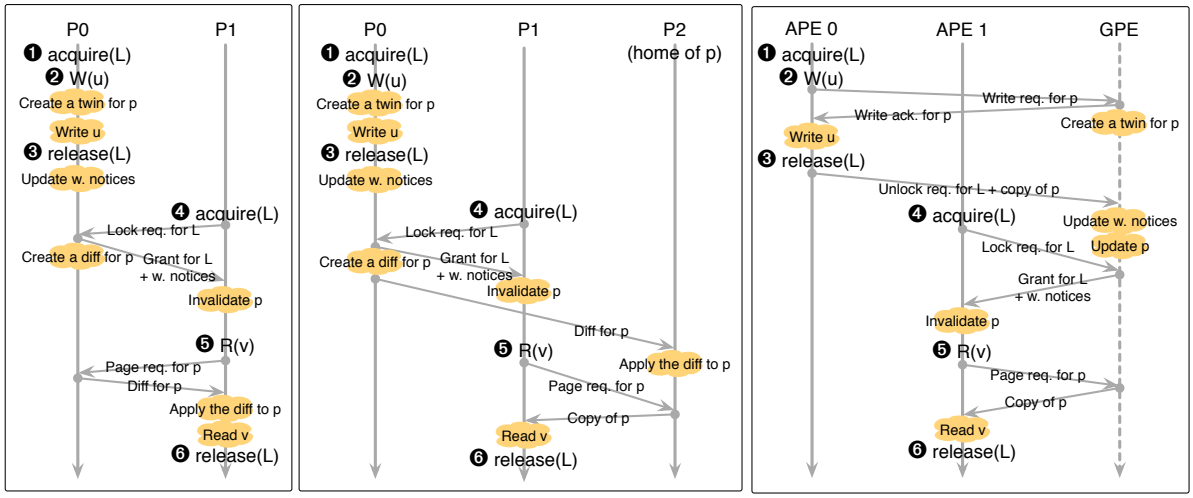
- We introduce a new memory consistency model, called hierarchical centralized release consistency (HCRC), and describe the protocol that makes it applicable across multiple levels of explicitly managed memory hierarchies.
- We describe the design and implementation of the COMIC++ software SVM that targets the heterogeneous multicore accelerator cluster. It is based on the HCRC protocol and software-managed caches.
- We show the effectiveness of COMIC++, evaluating its performance with a cluster system that consists of a manager node (a generic personal computer) and 32 Sony PlayStation3s connected by Gigabit Ethernet, using seven parallel benchmark applications.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 explains memory consistency models used in previous software SVM proposals that can be a basis to understand our SVM protocol. Section 4 presents the memory coherence and consistency protocol used in COMIC++. COMIC++ API functions are described in Section 5. Section 6 presents the results of evaluating the effectiveness of COMIC++. Finally, Section 7 concludes the paper.

2. Related Work

COMIC++ implements release consistency (RC)[11]. Important previous software SVM implementations of RC are eager release consistency (ERC)[6], lazy release consistency (LRC)[18, 19], home-based lazy release consistency (HLRC)[37], and centralized release consistency (CRC)[22]. These are closely related to COMIC++ and explained in the next section.

There exist many prior studies on page-based software SVM for loosely distributed memory multiprocessors including clusters of PCs or SMPs[2, 4, 6, 19, 18, 23, 29, 30, 33, 36, 37]. In the previous section, we differentiated COMIC++ from all these prior works, except Shasta[28, 29]. Shasta is a software-only approach that is similar to COMIC++. Shasta implements a shared address space by inserting inlined code in an executable at every load and store. The inlined code checks if the data is available locally. COMIC++ also requires replacing shared variable accesses with a call to COMIC++ API functions, and these calls are inlined in the application by a compiler.



(a) LRC (b) HLRC (c) CRC

Figure 3. Comparison between LRC, HLRC and CRC.

However, COMIC++ targets multiple levels of explicitly managed memory hierarchies.

Fatahalian *et al.*[10] design Sequoia, a language that incorporates the notion of hierarchical memory, and exposes communication and locality in the program. Bilas *et al.*[4] propose a method that exploits network processors to avoid asynchronous message processing in SVM systems. They modify the HLRC protocol to decouple protocol processing and message handling. The asynchronous message handling occurs only at the network interface (i.e., network processor), eliminating the need for interrupting the receiving host processor. In contrast, COMIC++ uses a coherence protocol that avoids sending asynchronous messages to the accelerator cores, and message handling is synchronous on the APE side by means of a three-way hand shaking protocol[22].

3. Background

In this section, we describe the memory consistency models for previous software SVM proposals, such as eager release consistency (ERC), lazy release consistency (LRC), home-based LRC (HLRC), and centralized release consistency (CRC). This will help readers understand the memory consistency model defined for COMIC++.

3.1. ERC, LRC, and HLRC

Memory consistency models for previous software SVMs are based on either an invalidate protocol or an update protocol. Since update based protocols are quite expensive due to heavy data transfers between nodes, they may not be appropriate for software SVMs[23]. An indication of coherence actions such as invalidations or updates is called a *write notice*. After receiving the write notices, a processor invalidates or updates all pages according to them.

LRC[7, 18, 19] is a well-known memory coherence and consistency model for software SVMs that targets traditional distributed memory multiprocessors. It is an implementation of RC[7, 11] and typically based on distributed algorithms to manage synchronization and page-level coherence. It alleviates the high frequency of coherence messages and data transfers that occur in the strict implementations of release consistency (RC), such as ERC[6].

ERC[6] is a precursor of LRC and another implementation of RC for software SVMs. Under ERC, a processor sends write notices to other processors at the release point. It may send write notices to more processors than necessary because it does not know which one is the next acquiring processor. On the other hand, LRC propagates write notices not at the release point, but at the time of an acquire by another processor. Write notices are sent to the acquiring processor only for the writes that precede the acquire. ERC has more coherence messages than LRC.

Since write notices under LRC flow from the previous releaser to the current acquirer of the same lock and accumulate in each processor, the acquirer needs to filter the write notices that have been performed with respect to itself. This requires establishing the happens-before relationship[1] between writes and synchronization operations due to causality. LRC divides the execution of each processor into distinct intervals[18, 19]. A new interval in a processor is delimited by a synchronization operation (i.e., an acquire or a release) executed by the processor. An interval is performed with respect to a processor if all writes in the interval have been performed with respect to that processor. On an acquire operation for a location, the acquirer sends its current vector time stamp to the previous releaser of the location. By comparing the received vector time stamp to its own, the releaser sends the acquirer the write notices for all intervals that have been performed with respect to the releaser but have not yet been performed with respect to the acquirer.

LRC supports multiple-writer protocols under which multiple processors can write to different locations of the same page simultaneously without any synchronization. This prevents the page from ping-ponging back and forth between different writers due to false sharing. Before a processor first writes a page within an interval, it creates a copy of the page, called *twin*. At the end of the interval, the processor compares the updated page with the twin and records differences in a structure called *diff*. It also creates a record of the write notices for the interval.

The first access to an invalidated page makes the processor collect all the diffs for the page from previous writers. These diffs are applied to the invalidated page in a proper

Table 1. Comparison between Different Memory Consistency Models

	ERC	LRC	HLRC	CRC
Applicability to the accelerator multicore architecture with local memory	No	No	No	Yes
Distributed or centralized algorithm	Distributed	Distributed	Distributed	Centralized
Avoiding asynchronous communication	No	No	No	Yes (no at the GPE)
Amount of protocol traffic	High	Medium	Low	Low (high at the GPE)
Amount of page traffic	High	Medium	Low	Low (high at the GPE)
Synchronization interval maintenance	No	Yes	Yes	No
Memory requirement	Low	High	Medium	Very low (high at the GPE)

causal order to construct the coherent page. Diffs can be created lazily on demand. Since the data structures for diffs and write notices consume a lot of memory and their life time is long, frequent garbage collection is needed[18] in LRC.

Figure 3 (a) illustrates a feasible scenario under LRC. At the beginning, all processors have a copy of page p for read in which variables `u` and `v` are located. Processor P0 has performed `acquire(L)` (❶). To execute `W(u)` (❷), P0 creates a twin for p and then writes `u`. Then it performs `release(L)` (❸). Since there was a write (i.e., `W(u)`) to p, P0 updates write notices that are sent to other processors later. P1 executes `acquire(L)` (❹). It sends a lock request to P0 that is the last owner of lock `L`. After receiving the lock request, P0 replies to P1 with a lock grant message for `L` that is piggybacked with write notices. Then, P0 creates a diff for p by comparing the updated copy of p with its twin. After receiving the write notices, P1 invalidates p in its memory. Now, P1 executes `R(v)` (❺). Since p has been invalidated, P1 sends to P0 a page request for p. P0 replies to P1 with the diff of p. After receiving the diff, P1 applies it to the invalidated copy of p in its memory, and then reads `v`.

HLRC[37] is a variation of LRC. In HLRC, a home location is maintained for each page, and the homes are distributed over multiple nodes. All updates to a page are propagated to its home and all copies are derived from the home. At the end of each interval, the writers compute diffs for updated pages and transfer them to their homes. Then, the writers discard the diffs. Home nodes apply the arriving diffs to their own copies as soon as they arrive and discard them. HLRC has several advantages over LRC. First, there are no page faults when accessing a page at its home node. Second, a non-home node can fetch the up-to-date page with a single round-trip communication without collecting diffs from the writers. Finally, protocol data structures consume less memory than LRC because the lifetime of diffs is extremely short.

Figure 3 (b) illustrates the same feasible scenario as that in (a) under HLRC. It is similar to the case in Figure 3 (a). But, after creating the diff, P0 sends it to P2 that is the home of p. After receiving the diff from P0, P2 applies the diff to p and updates p. Since p has been invalidated, P1 sends to P2 (the home of p) a page request for p when P1 executes `R(v)` (❺). P2 replies to P1 with a copy of p.

3.2. CRC

Even though the organization of the heterogeneous multicore processor in our cluster node is similar to that of traditional distributed memory multiprocessors, LRC or HLRC is not appropriate for the software SVM running on a multicore with explicitly managed memory hierarchies. First, the LRC protocol implemented in the heterogeneous multicore processor requires the APEs in the multicore to receive asynchronous messages during their computation for a task. Processing asynchronous messages will cause the APE to digress from performing their own computational task, and will degrade performance. Second, creating twins and diffs at the APE side also makes it digress from performing its computational task. Third, we cannot exploit an MMU for page fault detection and handling in APEs because they typically do not have such MMU functionality. Finally, since the APE local memory is typically small and there is no virtual memory support for the APEs, we face the problem of handling APE local memory overflows. For the same reason, creating twins and diffs at the APE side (i.e., the writer side under LRC or HLRC) consumes valuable APE local memory space.

CRC[22] is a variation of HLRC and solves these problems for a heterogeneous multicore with explicitly managed memory. The key idea in CRC is that managing synchronization and page-level coherence and consistency is centralized in a single PE in the heterogeneous multicore, i.e., in the GPE. The interconnect in a heterogeneous multicore typically has a way to guarantee in-order delivery of messages from the same APE to the GPE (e.g., ordered DMA operations). Thus, sequential consistency between synchronization operations can be easily guaranteed in CRC. No coherence action is initiated by the GPE, and therefore no asynchronous messages are sent to the APEs. Similar to HLRC, all pages have a single home, which is the GPE (the main memory). Moreover, twins are created at the GPE side (i.e., in main memory) that has more memory space than APEs. Managing APE local memory overflow is facilitated by a software-managed cache[9, 12, 32] that is implemented in the APE local memory. Page faults (i.e., misses in the software cache) are also detected by the software-managed cache.

When a write fault occurs in an APE (including the case where the APE has the page for read in its local memory), the APE sends a write request for the page to the GPE. After receiving the request, the GPE creates a twin of the faulting page in main memory, and if the APE does not already have the page for read, it sends a copy of the page to the APE. At the next release point, the APE sends the entire updated page to the GPE instead of creating a diff and sending it. The GPE compares it with the twin and constructs a coherent page in main memory.

The GPE maintains write notices for the updated pages with read and write requests from APEs. On the acquire request of an APE, the GPE replies to the acquirer with a lock grant message that is piggybacked with write notices. The APE invalidates all pages in its local memory (i.e., software cache) according to the write notices. When a page fault occurs after invalidation, the faulting APE fetches the up-to-date version of the page from the GPE (i.e., the main memory).

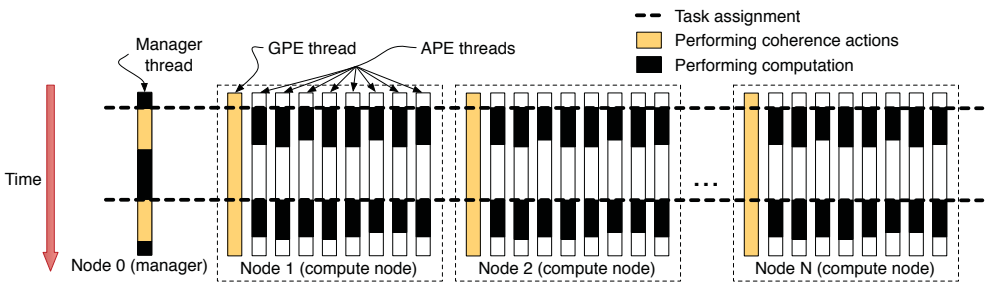


Figure 4. The thread model.

Figure 3 (c) illustrates CRC under the same scenario as that in Figure 3 (a). Since APE0 has p for read, APE0 sends a write request for p to the GPE to execute `W(u)` (❷). After receiving the request, GPE creates a twin for p and updates write notices because p will be updated by APE0. Then GPE replies to APE0 with a write acknowledgement. After receiving the write ack., APE0 writes `u`. Then APE0 performs `release(L)` (❸). Since there was a write (i.e., `W(u)`) to p, APE0 sends to GPE an unlock request that is piggybacked with an updated copy of p. After receiving the request, GPE updates p by comparing the received copy of p to its twin. Then, APE1 executes `acquire(L)` (❹). It sends a lock request to GPE. After receiving the lock request, GPE replies to APE1 with a lock grant message for `L` that is piggybacked with write notices. After receiving the lock grant message, APE1 invalidates p in its local memory. When APE1 executes `R(v)` (❺), GPE supplies a copy of p to APE1.

When the APE local memory overflows (i.e., the software cache is full), updated (i.e., dirty) pages in the local memory are sent to the GPE to make space available. After receiving the overflowed pages from an APE, the GPE handles those pages in the same way as it handles dirty pages due to a release. This implies that the GPE makes the updates in the dirty page visible to other APEs earlier than the next release point of the APE. However, this still satisfies the conditions of guaranteeing RC.

Since diff creation on the APE adversely affects performance by stealing computing cycles from the APE, the entire dirty page is sent to the GPE and the GPE performs the comparison with the twin. Moreover, there is no need to maintain intervals and vector timestamps at each APE for a proper causal order of events because managing locks and write notices is centralized in the GPE. The GPE can easily obtain the happens-before relationship between different writes and synchronization operations.

Table 1 summarizes the difference between different memory consistency models for previous software SVMs.

4. Memory Consistency in COMIC++

In this section, we describe the programming and memory consistency models on which COMIC++ is based. The memory consistency model is called *hierarchical CRC* (HCRC). We use a *global address* to refer to an address in the global memory, a *node local address* to refer to an address in the compute node local memory, and an *APE local address* to refer to an address in the local memory of an APE.

4.1. Programming model

The programming model for COMIC++ is an SPMD-like shared memory parallel programming model[8] in which the processor in the manager node and all the APEs in the cluster share a single global address space. Only the processor in the manager node and APEs are visible to the programmer. Similar to the case of ACMPs, the manager node executes serial portions of the application and the APEs execute parallel portions to boost performance.

The GPE in each compute node is not visible to the programmer, but it runs a thread to perform coherence actions. Application threads execute on the manager node and the APEs, one thread per PE. Writes to the shared address space by one thread are visible to reads of other threads according to release consistency (RC)[7, 11]. The thread running on the manager node (manager thread) performs sequential computation that is specified in the application and is responsible for assigning tasks to the compute nodes. It is also responsible for maintaining coherence between the compute nodes (i.e., at the node local memory level) and managing synchronization between them. The GPE thread in each compute node is responsible for assigning tasks and providing OS services to each APE in the same node. It also manages synchronization between APE threads in the same node. The GPE thread follows a round-robin policy to service APE threads to guarantee fairness and to avoid starvation. The same is true between the manager thread and the compute nodes.

Figure 4 illustrates our thread model. When the system starts, the manager, GPE, and APE threads are all created. The manager thread notifies the GPE threads when to assign tasks to APE threads. The GPE thread in a compute node assigns tasks to the APE threads in the same node. When an APE thread is created, it waits for task assignment from the GPE thread. After performing the assigned task, the APE again waits for more tasks to be assigned by the GPE thread. In turn, the GPE thread waits for more tasks to be assigned by the manager thread.

4.2. HCRC

Unlike previous memory consistency proposals for software SVMs, HCRC is applicable across multiple levels in the explicitly managed memory hierarchy. HCRC guarantees coherence and consistency with CRC between APEs in the same node (i.e., across the different APE local memories) and between the APE local memory and the node local memory. HCRC guarantees coherence and consistency between different nodes (i.e., across different node local memories) with a protocol similar to HLRC. Each page has a single home node and homes are distributed over the man-

ager and compute nodes. Each compute node (i.e., a GPE) implements a page table for all pages and a page buffer in the node local memory in which to cache pages. The GPE in each compute node relays synchronization requests from the APEs in the same node to the manager node, and relays page requests from the APEs to the home nodes of the pages.

All acquire and release synchronization requests from an APE are relayed by the GPE in the same node to the manager node. The manager node processes the synchronization requests in the order they arrive and in a round-robin manner for different nodes. CRC preserves the program order between synchronization requests in the same node, and the requesting GPE gives an incremental identifier to each synchronization request when it sends the request to the manager node. Thus, the program order of synchronization operations can be preserved at the manager node by exploiting the identifier even though the interconnection network does not guarantee in-order delivery of messages. Hence, sequential consistency between synchronization operations can be easily guaranteed.

When there is a page request for page p from an APE to the GPE in the same node, and it is a hit in the page buffer, the APE and the GPE just follow the CRC protocol. When it is a miss, the GPE relays the request to the home node:

- Read miss. When p is not in the page buffer, the GPE sends a read request to the home node of p. The home node replies to the GPE with a copy of p. In addition, the home node sends a read acknowledgement message to the manager node. The manager node uses this information to construct write notices. After receiving the copy of p, the GPE caches it in the page buffer. Sending the copy to the requester APE follows the CRC protocol.
- Write miss. When p is not in the page buffer, the GPE sends a write request to the home node of p. The home node replies to the GPE with a copy of p. In addition, the home node sends a write acknowledgement message to the manager node. After receiving the copy, the GPE caches it in the page buffer and creates a twin. Sending the copy to the requester APE follows the CRC protocol. The case when p is in the page buffer for read is treated as a write miss, but instead of replying with a copy of p, the home node replies to the GPE with a write acknowledgement message.

When the GPE relays a release for a location `L` from an APE in the same node, the GPE sends an unlock request that is piggybacked with the updated page list for `L` to the manager node. After receiving the unlock request, the manager node updates write notices. The GPE also creates diffs by comparing the updated copies in the page buffer with their twins. The GPE sends the diffs to the homes of the updated pages. After receiving these diffs, the home nodes send update acknowledgement messages to the manager node. At the same time, the home nodes apply the diffs to their original copies and update the pages. After receiving all update acknowledgement messages from the homes of updated pages for `L`, the manager node can grant `L` again.

When the GPE relays an acquire for a location `L`, it sends a lock request to the manager node. After receiving the request, the manager node replies to the GPE with a lock grant message that is piggybacked with write notices when `L` is available. After receiving the lock grant message, the GPE invalidates pages in the page buffer according to the write notices.

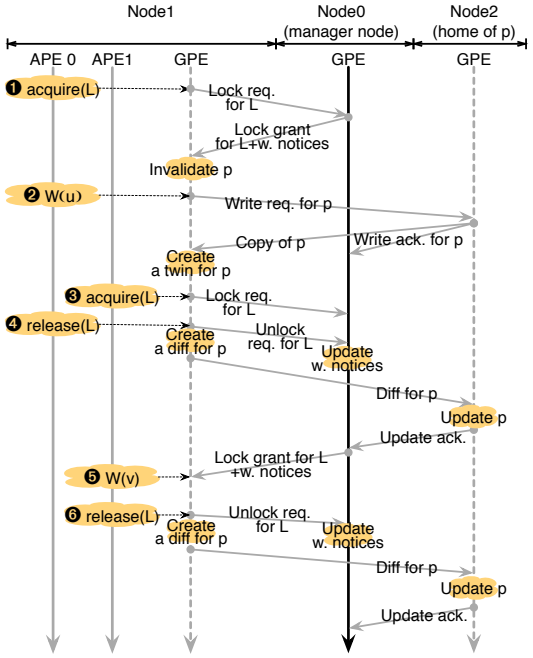


Figure 5. A scenario under HCRC. Variables `u` and `v` are located in page p. Node0 is the manager node and Node2 is the home of p.

Figure 5 illustrates a feasible scenario under our HCRC implementation. At the beginning, Node1 has p, but APE0 and APE1 do not have p in their local memory. Page p has been updated before in its home node, but the copy of p in Node1 has not been updated yet. APE0 performs `acquire(L)` (❶). The GPE at Node1 relays this request to Node0 (i.e., to the GPE at the manager node). The manager replies to Node1 with a lock grant message piggybacked with write notices that indicate p has been updated. Thus, Node1 (i.e., the GPE) invalidates p in the node local memory. APE0 executes `W(u)` (❷). The GPE at Node1 relays the write request from APE0 to Node2 (the home of p) and Node2 replies to Node1 with a copy of p. Simultaneously, Node2 sends a write ack. message for p to Node0. Node0 uses this information to construct write notices for each node in the cluster. After receiving a copy of p, Node1 creates a twin for p and replies to APE0 with a copy of p. APE1 performs `acquire(L)` (❸). The GPE at Node1 relays this request to Node0. Since APE0 already owns `L`, Node0 does not reply to Node1 with a lock grant message for `L`. This makes APE1 wait for `L` until it receives a lock grant message. Then, APE0 performs `release(L)` (❹). After receiving the unlock request and the updated copy of p from APE0, the GPE at Node1 creates a diff for p by comparing the copy from APE0 with the twin of p. Simultaneously, the GPE relays the unlock request to Node0. After receiving the unlock request, Node0 updates write notices. The diff is sent to Node2. After receiving the diff, Node2 applies it to

its own copy of p. Then, Node2 sends to Node0 an update ack. that tells Node0 that updating p has been finished at p's home. After receiving the update ack., Node0 replies to Node1 with a lock grant message for `L` that is piggybacked by write notices. Since Node1 itself is the previous writer of p, the write notices do not contain any information about the updates to p. APE1 performs `W(v)` (❺). Since Node1 has p in the node local memory, the GPE supplies it to APE1. Then, APE1 performs `release(L)` (❻). This case is similar to the case of ❹, and the copy of p at the home node is updated.

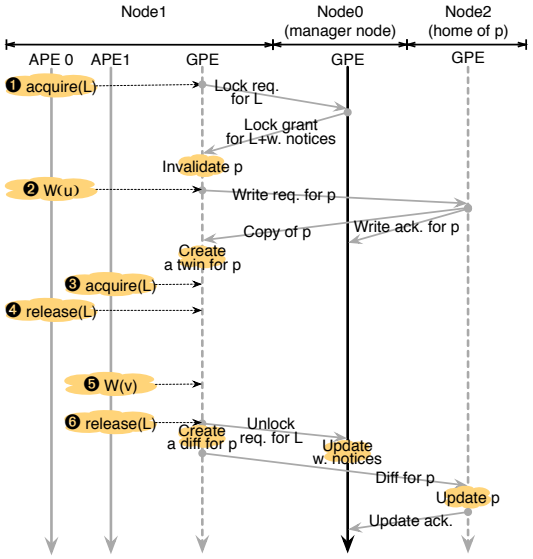


Figure 6. Exploiting locality of locks in HCRC.

4.3. Exploiting locality of locks

APEs in the same compute node typically contend for the same lock at the same time. We can exploit this property to reduce message and data traffic between nodes. When an acquire operation for `L` is executed by an APE, say APE1, before a release operation for `L` is performed by another APE in the same node, say APE0, the release operation by APE0 can be combined with the acquire operation to be performed by APE1. This case is illustrated in Figure 6. The message and data traffic due to release operations at the node level is significantly reduced when compared to the case in Figure 5. APE0 and APE1 are contending for `L` in Node1. When APE0 executes `release(L)` (❹), the GPE checks whether there is a lock request for `L` from another APE. If so, the GPE does not relay the release operation. Instead, locks are managed locally by the GPE. When APE1 executes `release(L)` (❻), `L` is released outside of Node1 and the diffs of updated pages in Node1 are sent to their home nodes because there are no pending acquire operations.

In HCRC, when the GPE in a compute node observes cascaded and overlapped acquire-release pairs for the same lock from APEs, the GPE does not relay the release operation in the pairs until there is no pending acquire operation in the node, or the count of acquires reaches a certain threshold value. Typically this threshold value is given by $c \cdot N$ to guarantee fairness between nodes and to avoid starvation due to synchronization, where c is a constant and N is the number of APEs in a compute node. In other words, lock management is local to the compute node (i.e., the GPE) for $c \cdot N$ consecutive lock requests in the node. This significantly reduces message and data traffic between nodes.

4.4. Data structures to implement HCRC

The runtime system implements a page table for all shared-data pages in the manager node. Each page table entry contains the home node location, and the read/write status of each compute node for the corresponding page. Whenever there is an unlock request from a compute node, the manager node builds appropriate write notices using the page table in combination with lock owner information maintained at run time.

Each compute node has a page buffer to cache pages at the node local memory level. It also has a page table that is similar to the one in the manager node. Each shared-data page has an entry in this table. With the page table, the GPE in the compute node maintains the following information for pages: validity in the page buffer, location in the page buffer, home node, and read/write status. The GPE maintains another page table, called APE page table, for the APEs within the same node. Each APE page table entry records the read/write status of each APE for the corresponding page, and the location of the page in the page buffer. Whenever there is an unlock request from an APE, the GPE builds appropriate write notices that are sent to the APEs using the APE page table and lock owner information maintained at run time.

Each APE has a software-managed cache, such as ESC[32], implemented in its local memory. Pages are cached in the software-managed cache, and misses in the cache cause page requests to the GPE in the same node. Since COMIC++ does not exploit MMU's page fault handling mechanism at all, the page size need not be the same as the page size in the OS. In addition, the page size in the cache (i.e., the size of pages for the APE page table in the node local memory) and the page size in the page buffer in the node local memory need not be the same. However, the page size at the node local memory level should be greater than or equal to the page size at the APE local memory level to reduce message and data traffic.

Since virtual memory support is provided by the OS for the node local memory, the total size of the page tables and page buffer can be as large as the virtual address space. However, to avoid performance degradation due to swapping, it is better to keep the page buffer size less than the size of available physical node local memory. We can apply an appropriate page replacement policy to the page buffer. When it is full, a victim is selected and replaced. We do not replace the page whose home is the compute node itself.

4.5. Home node selection

A clever assignment of home nodes can significantly reduce message and data traffic. In COMIC++, the home node for a page p is chosen by the following rule. *The compute node that first writes a page becomes the home of the page until execution finishes.* Initially, the manager node has all shared-data pages in the application. Note that the OS in the manager node provides virtual memory support. Each compute node and the manager node record the home of each page in their page table. Initially, no page has a home. If there is a write request and there is no home

`#define COMIC_PP_shared( X )` A macro. Shared variable declarations are passed to the macro as an argument.
`#define COMIC_PP_access( X )` A macro. Each shared variable access in the manager thread is marked by this macro.
`#define COMIC_PP_MAX_NUM_APE_THREADS` The maximum number of APE threads supported by COMIC++.
`void COMIC_PP_manager_init();` Initialize the COMIC++ runtime system and create APE threads.
`void COMIC_PP_manager_exit();` Terminate all APE threads and exit.
`void COMIC_PP_manager_run( unsigned int func_id );` Assign the function whose ID is `func_id` to all APE threads.

Figure 7. COMIC++ API functions for the manager thread.

`#define COMIC_PP_shared( X )` A macro. Shared variable declarations are passed to the macro as an argument.
`#define COMIC_PP_access( X )` A macro. Each shared variable access in an APE thread is marked by this macro.
`#define COMIC_PP_ape_main( X )` A macro. Case statements, each of which contains a task, are passed to the macro as an argument.
`#define COMIC_PP_MAX_NUM_APE_THREADS` The maximum number of APE threads supported by COMIC++.
`unsigned COMIC_PP_get_ape_thread_id();` Obtain the ID of the calling APE thread.
`void COMIC_PP_ape_barrier();` Block the calling thread until all other threads arrive at the barrier.
`void COMIC_PP_ape_lock_acquire( unsigned lock );` Block the calling thread until it acquires the lock.
`void COMIC_PP_ape_lock_release( unsigned lock );` Release the specified lock.
`void COMIC_PP_ape_flush_all();` Flush to the main memory and invalidate all modified local copies of the pages.
`void COMIC_PP_ape_flush( void *ptr );` Flush to the main memory and invalidate the modified local copy of the page that contain the shared address `ptr`.
`void *COMIC_PP_ape_local_address( void *ptr );` Return the address in the APE local memory that maps to the shared address `ptr`.
`void *COMIC_PP_ape_malloc( unsigned size );` Allocate `size` bytes in the shared address space.
`void COMIC_PP_ape_free( void *ptr );` Frees releases the memory pointed to by ptr.
type `COMIC_PP_ape_read_char(` *type* `*ptr );` Return the value of each variable at the shared address `ptr`. *type* is one of `char`, `short`, `int`, `long`, `long long`, `float`, and `double`.
type `COMIC_PP_ape_write_char(` *type* `*ptr,` *type* `v );` Write the value of variable `v` to the shared address `ptr`. *type* is one of `char`, `short`, `int`, `long`, `long long`, `float`, and `double`.

Figure 8. COMIC++ API functions for APE threads.

for the page, the manager node records the requester as the home of the page in its page table. When a compute node accesses a page that has no home recorded in its page table, the node sends a request for the page to the manager node. The manager checks its table and informs the requester where the home is. Then the requester records the home in its page table and obtains the page from the home. If there is no home recorded in the manager's page table and the request is a read, the manager replies to the requester with the page.

5. Application Programming Interface

The COMIC++ API provides functions for reading and writing shared variables, thread creation, destruction, and synchronization. Figure 7 lists C API functions for the manager thread, and Figure 8 for APE threads.

```
#include <omp.h>
#define SIZE 1024

int a[ SIZE ];

void main()
{
  int i;

#pragma omp for
  for(i=0; i<SIZE; i++)
    a[i] = i;
}
```

(a)

```
#include <comic_pp_manager.h>
#define SIZE 1024

COMIC_PP_shared (
  int a[ SIZE ];
)

void main()
{
  COMIC_PP_manager_init();
  COMIC_PP_manager_run(0);
  COMIC_PP_manager_exit();
}
```

(b)

```
#include <comic_pp_ape.h>
#define SIZE 1024

COMIC_PP_shared (
int a[SIZE];
)

COMIC_PP_ape_main (
  case 0:
    {
      int i, tid, chunk, low, high;

      tid = COMIC_PP_get_ape_thead_id();
      chunk = SIZE / COMIC_PP_MAX_NUM_APE_THREADS;
      low = tid * chunk;
      high = (tid == (COMIC_PP_MAX_NUM_APE_THREADS-1))
               ? SIZE : (low + chunk);

      for(i=low; i<high; i++)
        COMIC_PP_ape_write_int(&COMIC_PP_access(a[i]), i);

      COMIC_PP_ape_barrier();
    }
    break;
)
```

(c)

Figure 9. (a) An OpenMP program. (b) COMIC++ program for the manager thread. (c) COMIC++ program for APE threads.

A sample COMIC++ parallel program is shown in Figure 9 (b) and (c). Figure 9 (a) is an OpenMP parallel program in which multiple threads can execute different iterations of the `for` loop in parallel. The OpenMP program can be converted to two C programs that contain COMIC++ API functions.

Figure 9 (b) shows the manager program that becomes the manager thread. Array `a` is declared as a global shared array. The manager thread creates APE threads (`COMIC_PP_manager_init();`) and assigns a task (the task ID is equal to 0) to each APE thread (`COMIC_PP_manager_run(0);`). Each thread executes the same SPMD-style code fragment that corresponds to the task ID. Then, the manager thread exits (`COMIC_PP_manager_exit();`).

Figure 9 (c) is the SPMD-style APE program that each APE thread executes. The main function in the APE program (not included in Figure 9 (c)) loops until the manager thread assigns a task. The case statement that matches the task ID is executed after a task is assigned. After finishing the task, the APE thread loops again. Each thread is given a chunk of iterations of the parallel loop. Since there is an implicit barrier at the end of a parallel `for` loop in OpenMP, a COMIC++ barrier is inserted at the end of the loop.

6. Performance

This section presents the evaluation environment and performance results for the current implementation of COMIC++.

Table 2. System Configurations

For evaluating the SVM on a single multicore (only 6 SPEs are used)	
IBM QS 21 Cell Blade Server Two 3.2GHz Cell BE processor, 8 SPEs each 512KB L2 cache per processor, 2GB main memory, Fedora Linux 9	
Heterogeneous multicore accelerator cluster	
Manager node (Power Station)	Two 2.5GHz IBM970MP dual-core CPUs (Only one core is used for the node) 4GB main memory, Yellow Dog Linux 6.1
Compute node (Sony PS3)	3.2GHz Cell BE processor, 6 SPEs 256MB main memory, Yellow Dog Linux 6.1
Network switch	3COM 48-port Gigabit Ethernet switch
Compilers used	
IBM XL Cell ppuxlc, IBM XL Cell spuxlc, IBM XL Cell ppu32-embedspu (embedder)	

6.1. Evaluation environment

We evaluate the performance of COMIC++ using a single Cell BE processor with 6 SPEs and an accelerator cluster that consists of a workstation and 32 Sony PlayStation3s. The Cell BE heterogeneous multicore processor consists of a single Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE has two levels of cache hierarchy that are coherent with the main memory, but each SPE has 256KB of local memory without any cache. An SPE can transfer data from/to the external main memory using an asynchronous DMA operation, and it supports 128-bit SIMD instructions. Table 2 summarizes the parameters of the target machines and compilers used in our evaluation.

The COMIC++ runtime is implemented using GASNet library[5] and the IBM Cell SDK library[15]. GASNet is a low-level networking layer that provides high-performance communication primitives.

Table 3. Applications Used

App.	Source	Input	Shared Data
Blackscholes	PARSEC[3]	22000000 options, 3000 iters.	1.28GB
EP	NAS[25]	class C	15.7KB
Jacobi	OpenMP[27]	5000 × 5000, 2000 iters.	572.2MB
LU	SPLASH-2[35]	6000 × 6000	275.2MB
MD	OpenMP[27]	65536 particles	6.0MB
Raytrace	SPLASH-2[35]	balls4 with antialiasing level 170	32.3MB
Water-Sp	SPLASH-2[35]	474552 molecules	308.2MB

We run seven shared memory applications on COMIC++ that are from various sources. Their properties and input data sets are summarized in Table 3. To generate parallel programs for COMIC++, we convert the applications manually to programs using the COMIC++ API functions. Each shared memory access is replaced with a proper COMIC++ API function to access software caches implemented in the SPE local store. It is possible to automate this translation process using a compiler. We faithfully follow the semantics of original parallel programs in our manual translation.

Table 4. Costs of Primitive Operations

Communication		
Message size	Mailbox (between GPE and SPE)	GASNet short message (PS3 to PS3)
4B	136.2 ns	148.3 μs
Data Transfer		
Data size	DMA (between GPE and SPE)	GASNet long message (PS3 to PS3)
2KB	171.6 ns	201.4 μs
4KB	305.2 ns	313.4 μs
8KB	688.6 ns	332.7 μs
16KB	1.373 μs	432.9 μs

6.2. Costs of primitive operations

Table 4 shows latencies of primitive communication and data transfer operations. COMIC++ implements the basic communication mechanism in a node with mailboxes (supported in Cell BE hardware), and across nodes with GASNet short messages. It implements intra-node data transfers with DMA operations and inter-node data transfers with GASNet long messages. The intra-node communication or data transfer cost is three orders of magnitude cheaper than that of inter-node communication or data transfer.

6.3. Speedup on the single multicore processor

Figure 10 shows the performance of COMIC++ on a single Cell BE processor with six SPEs. In this case, COMIC++ implements only the CRC protocol. We vary the page size from 2KB to 16KB. 16KB is the maximum amount of data that can be transferred with a single DMA operation. Y-axis shows the speedup that is obtained over a single PPE core. Since the speedup of Blackscholes is much bigger than that of others, we divide the graph into two sections with different scales. The optimal page size varies for different applications.

Table 5 shows statistics for each application for the page size that gives the best performance. It shows the page read and write miss rates, the read-to-write upgrade rate, the number of write backs due to cache misses and synchronization per SPE, the number of barriers executed, and the total number of lock requests from the six SPEs.

Blackscholes achieves the best speedup. One of the reasons for this is that the SPE is specialized to accelerate single-precision floating-point operations, and Blackscholes is a single-precision floating-point application. In addition, there is no writeback due to misses or synchronization. It has a high read miss rate, but it does not write to shared data at all. The protocol overhead incurred by read misses is relatively smaller than that incurred by write misses. Although EP has a write miss rate higher than that of others, EP has a very small number of reads and writes. This results in almost no protocol and page traffic for EP, and it achieves better speedup than all other applications except Blackscholes and Raytrace.

All applications except Blackscholes and EP have heavy protocol and page traffic due to the high page miss rate, the large number of write backs, or many lock requests. LU and Water-Sp achieve relatively bad speedup due to

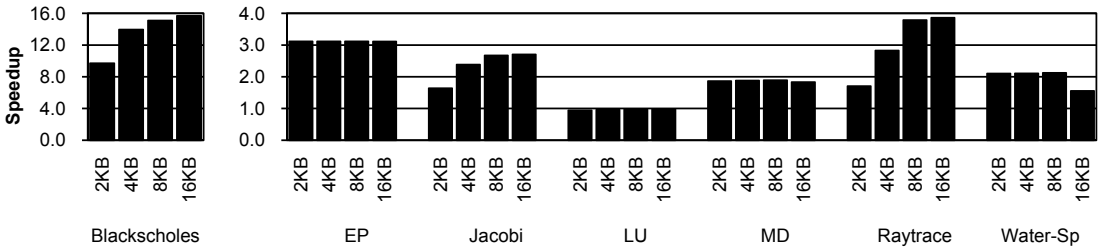


Figure 10. Speedup on a single Cell BE processor with 6 SPEs. The page size varies from 2KB to 16KB.

Table 5. Statistics of Each Application on a Single Cell BE processor with 6 SPEs

	Page size	Read miss rate	Write miss rate	R→W upgrade rate	WB due to misses	WB due to Sync	Barrier	Total lock requests
Blackscholes	16KB	1.024%	0.000%	0.000%	0	0	1	0
EP	16KB	0.178%	25.676%	0.000%	0	3	3	6
Jacobi	16KB	0.666%	0.216%	0.000%	8,120,000	19,000	4,001	0
LU	16KB	0.008%	0.000%	0.003%	460,521	1,470	754	6
MD	8KB	1.121%	0.047%	0.000%	1,004,812	208	21	0
Raytrace	16KB	0.647%	2.884%	12.549%	2,027,387	1,163	2	2,048
Water-Sp	8KB	0.105%	0.001%	0.867%	1,240,454	51	21	115

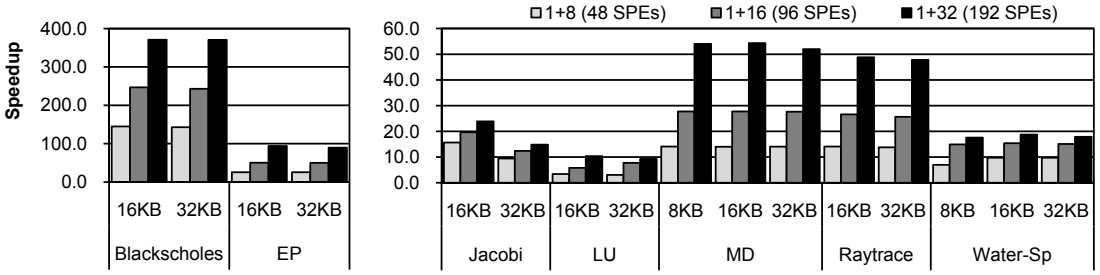


Figure 11. Speedup on the cluster with various inter-node page sizes.

Table 6. Statistics of Each Application on the Cluster (page size=16KB, 192 SPEs)

	Read miss rate (APE)	Write miss rate (APE)	R→W upg. rate (APE)	Read miss rate (node)	Write miss rate (node)	R→W upg. rate (node)	WB due to misses (node)	WB due to Sync (node)	Barrier	Total lock requests		Locks handled locally
										Global	Local	
Blackscholes	1.059%	0.000%	0.000%	0.032%	0.000%	0.000%	0	0	1	0	0	-
EP	0.194%	27.294%	0.000%	56.919%	92.721%	0.000%	0	1	3	152	192	20.8%
Jacobi	0.677%	0.217%	0.000%	0.397%	0.151%	0.000%	0	323	4,001	0	0	-
LU	0.023%	0.000%	0.006%	0.993%	0.000%	0.549%	0	35	754	152	192	20.8%
MD	1.123%	0.047%	0.000%	0.021%	0.010%	0.000%	0	0	21	0	0	-
Raytrace	0.642%	2.903%	12.551%	0.002%	0.135%	0.072%	0	42	2	3,986	3,988	0.1%
Water-Sp	0.058%	0.001%	0.397%	0.835%	16.953%	0.671%	0	22	21	3,047	3,649	16.5%

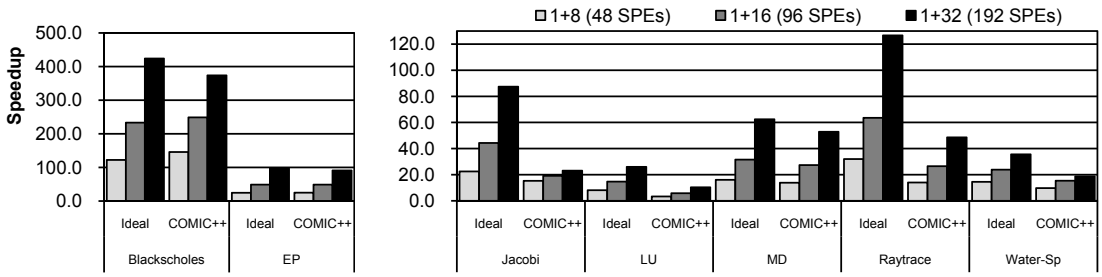


Figure 12. Scalability of each application (page size = 16KB).

many branches. The SPE is very sensitive to branches. It has no hardware branch predictor, and the branch penalty can sometimes be as high as twenty times the latency of non-branch instructions[16]. LU has many branches inside its parallel loops. Water-Sp has a few function calls (i.e., branches) inside its parallel loops, but their effect on performance is not that significant compared to the case of LU. Although the number of total lock requests in Raytrace is bigger than Water-Sp, the degree of lock contention for the same lock is almost zero. Most of the lock requests in Raytrace try to acquire different locks. In this case, the latency of acquiring a lock is a few hundred nano-seconds (time for a send and a receive through the mailbox, as shown in Table 4). Blackscholes and Jacobi contain no branches in their parallel loops, and EP and MD contain very few branches in their parallel loops.

6.4. Speedup on the cluster

Figure 11 shows the performance of COMIC++ on the target accelerator cluster. We set the intra-node page size, p, to be the one that achieves the best speedup on a single Cell BE processor and vary the inter-node page size from p to 32KB. 32KB is the maximum power-of-two page size that GASNet can send in a single message. We also vary the number of compute nodes in the cluster from eight (i.e., a total of 1+8 nodes) to thirty two (i.e., a total of 1+32 nodes). Y-axis shows the speedup that is obtained over a single PPE core. Again, we divide the graph into two sections with different scales because the speedup numbers for Blackscholes and EP are much higher than the numbers for other benchmarks. For all applications, the optimal inter-node page size is 16KB. For MD and Water-Sp, this size is different from their intra-node page size.

Assume that T_p is the sequential execution time of the portion of an application that can be made parallel and T_s is the sequential execution time of the entire application. Also assume that the speedup $S_p(n)$ of the parallel portion is linearly proportional to the number of SPEs n and $S_p(1) = 1$. The actual speedup of $S_p(6)$ with six SPEs can be obtained by running the parallel portion (i.e., the application) on a single Cell BE processor. Then, the speedup of the parallel portion with n SPEs can be estimated by,

$$S_p(n) = \frac{S_p(6)}{6} \cdot n + (1 - \frac{S_p(6)}{6})$$

This gives an estimation of the ideal speedup of the application on n SPEs without any overhead incurred by COMIC++ across nodes:

$$S_{ideal} = \frac{T_s}{(T_s - T_p) + T_p/S_p(n)}$$

Figure 12 shows S_{ideal} and the actual speedup of each application when the page size is 16KB and the number of SPEs is 48, 96, and 192. The difference between S_{ideal} and the actual speedup is the overhead incurred by COMIC++ due to the inter-node protocol processing and page traffic. The bigger this difference is, the worse is the scalability of the application. The shared data size in Blackscholes (1.28GB in Table 3) is bigger than the size of the main memory partition (1GB) that is allocated to a single Cell BE processor in our QS21 blade server. The shared data is distributed over two different partitions (1GB each), and it takes more time for the processor to access the data in the other partition. This makes $S_p(6)$ worse than that with the partition size big enough for the entire shared data. The cluster does not have this problem. This is the reason why S_{ideal} of Blackscholes is worse than the actual speedup obtained for 48 and 96 SPEs in Figure 12.

Table 6 shows the characteristics of each application for the inter-node page size that gives the best performance on 32 compute nodes (a total of 192 SPEs). In addition to intra-node, per-SPE statistics, it shows inter-node, per-node statistics: page read and write miss rates, read-to-write page upgrade rate, and the numbers of write backs due to misses and due to synchronization. It also shows the number of barriers executed and the total number of global lock requests (from a node) and local lock requests (from an SPE). The last column shows the ratio of the number of locally managed lock requests to the number of total local lock requests.

As shown in Table 4, the inter-node communication latency (a few hundred micro-seconds) is much bigger than the intra-node communication latency. Thus, an application may not scale well if it has bad inter-node statistics in Table 6. Jacobi, LU, Raytrace, and Water-Sp have such statistics and Figure 12 confirms that they do not scale well due to the inter-node protocol overhead and page traffic. MD achieves a relatively poor speedup on a single Cell BE processor, but it achieves good scalability on the accelerator cluster. Note that EP has a very small number of reads and writes to shared data and Raytrace has many lock requests that are handled globally.

For EP, LU and Water-Sp, more than 16% of the total lock requests are handled locally. This implies that the mechanism that exploits lock locality in a compute node is quite effective and significantly reduces inter-node communication traffic.

The experimental results indicate that the performance of an application on a single accelerator multicore with COMIC++ is determined not only by the protocol overhead and page traffic between different cores, but also by specific characteristics of the application itself. These characteristics determine the performance penalty incurred when porting the application code to the accelerator core. On the other hand, for the accelerator cluster, the protocol overhead and page traffic between different accelerator multicore processors dominate the performance and scalability of the application.

7. Conclusion

We introduce a software SVM system (COMIC++) for heterogeneous multicore accelerator clusters with explicitly managed memory hierarchies. Since coherence and consistency protocols of previous software SVM proposals cannot efficiently handle multiple levels of memories in an explicitly managed memory hierarchy, we propose a new coherence and consistency protocol, called hierarchical centralized release consistency (HCRC). To our knowledge, COMIC++ is the first SVM system that handles multiple levels of memories in an explicitly managed memory hierarchy.

An efficient software SVM is critical for programmability and performance of shared memory applications that are parallelized across a large number of cores. Together with the trend in high performance computing towards the use of accelerator cores, this implies that the software SVM needs to operate efficiently on a cluster of heterogeneous

accelerator processors and their distinct memory hierarchies. COMIC++ enables this using the HCRC protocol and software-managed caches in different memory levels. The HCRC protocol implements different protocols for different levels of the explicitly managed memory hierarchy.

Experimental results show that the hierarchical approach in COMIC++ works quite effectively with a heterogeneous multicore accelerator cluster. For performance on a single accelerator multicore running COMIC++, the sensitivity of an application to the computational specialty of the accelerator core is as important as the communication and data transfer overhead between cores. On the other hand, the inter-node communication and data transfer overhead dominates the performance and scalability of applications on the accelerator cluster running COMIC++. This is because the latency of communication and data transfer is three orders of magnitude higher for the accelerator cluster compared to a single multicore processor in our system.

References

[1] S. Adve and M. Hill. Weak Ordering. In *ISCA'90: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, September 1991.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, October 2008.

[4] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *ISCA '99: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 282–293, May 1999.

[5] D. Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, Department of EECS, University of California at Berkeley, October 1991.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP '91: Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[7] D. E. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[8] F. Darema. The SPMD Model: Past, Present and Future. *Lecture Notes in Computer Science*, 2131(1):1–1, January 2001.

[9] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine^{TM} architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[10] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 Supercomputing Conference*, November 2006.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[12] M. Gonzalez, N. Vujic, A. E. Eichenberger, T. Chen, X. Martorell, E. Ayguadé, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien. Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture. In *PACT08: Proceedings of the 17th ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques*, pages 303–314, October 2008.

[13] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March/April 2006.

[14] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, July 2008.

[15] IBM. *Software Development Kit for Multicore Acceleration version 3.0, Programmer's Guide*. IBM, 2007. `http://www.ibm.com/developerworks/power/cell/`.

[16] IBM. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*. IBM, 2008. `http://www.ibm.com/developerworks/power/cell/`.

[17] IBM, Sony, and Toshiba. *Cell Broadband Engine Architecture*. IBM, October 2007. `http://www.ibm.com/developerworks/power/cell/`.

[18] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–131, January 1994.

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA'92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[20] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton. Petascale Computing with Accelerators. In *PPoPP09: Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 241–250, February 2009.

[21] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *IEEE Computer*, 38(11):32–38, November 2005.

[22] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell BE. In *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–314, October 2008.

[23] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *PODC '86: Proceedings of the fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[24] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguadé. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Computer Architecture Letters*, 5(1):4, January 2006.

[25] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[26] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA, June 2008. `http://developer.download.nvidia.com`.

[27] OpenMP. OpenMP. `http://www.openmp.org`.

[28] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *HPCA '98: Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 125–136, January 1998.

[29] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: a Low overhead, Software-only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[30] 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 *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 297–306, October 1994.

[31] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[32] S. Seo, J. Lee, and Z. Sura. Design and Implementation of Software-Managed Caches for Multicores with Local Memory. In *HPCA'09: Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture*, pages 55–66, February 2009.

[33] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[34] Tilera Corporation. *TILEPro64 Multicore Processor*, 2009. `http://www.tilera.com`.

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

[36] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software write detection for distributed shared memory. In *OSDI '94: Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 87–100, November 1994.

[37] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *OSDI '96: Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88, October 1996.