COMIC: A Coherent Shared Memory Interface for Cell BE[*]

Jaejin Lee[†], Sangmin Seo[†], Chihun Kim[†], Junghyun Kim[†],
Posung Chun[†], Zehra Sura[‡], Jungwon Kim[†], and SangYong Han[†]

[†]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, {sangmin, chihun, junghyun, posung}@aces.snu.ac.kr,
zsura@us.ibm.com, jungwon@aces.snu.ac.kr, syhan@pandora.snu.ac.kr

http://aces.snu.ac.kr

ABSTRACT

The Cell BE processor is a heterogeneous multicore that contains one PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). Each SPE has a small software-managed local store. Applications must explicitly control all DMA transfers of code and data between the SPE local stores and the main memory, and they must perform any coherence actions required for data transferred. The need for explicit memory management, together with the limited size of the SPE local stores, makes it challenging to program the Cell BE and achieve high performance. In this paper, we present the design and implementation of our COMIC runtime system and its programming model. It provides the program with an illusion of a globally shared memory, in which the PPE and each of the SPEs can access any shared data item, without the programmer having to worry about where the data is, or how to obtain it. COMIC is implemented entirely in software with the aid of user-level libraries provided by the Cell SDK. For each read or write operation in SPE code, a COMIC runtime function is inserted to check whether the data is available in its local store, and to automatically fetch it if it is not. We propose a memory consistency model and a programming model for COMIC, in which the management of synchronization and coherence is centralized in the PPE. To characterize the effectiveness of the COMIC runtime system, we evaluate it with twelve OpenMP benchmark applications on a Cell BE system and an SMP-like homogeneous multicore (Xeon).

[*]This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software], by the IBM Corporation, and by the Ministry of Education, Science and Technology under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*hardware/software interfaces*; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; D.4.2 [**Operating Systems**]: Storage Management—*virtual memory*

General Terms

Algorithms, Design, Experimentation, Languages, Management, Measurement, Performance

Keywords

Software Distributed Shared Memory, Software Shared Virtual Memory, Heterogeneous Multicores, OpenMP, Cell BE

1. INTRODUCTION

Recently chip multiprocessors are widening their user base in all computing domains. Some examples are Intel Xeon, AMD Opteron, and IBM Power5 processors, in which the cores are homogeneous and the organization is similar to conventional symmetric multiprocessor (SMP) systems. An interesting and radically different architecture from the SMP approach is the Cell Broadband Engine (Cell BE) designed by Sony, Toshiba, and IBM [15, 18, 27]. It is a heterogeneous multicore capable of massive floating-point processing for compute-intensive workloads. It is produced in high volumes and used in the Sony PlayStation 3 gaming systems, as well as in IBM Cell blade servers.

The Cell BE consists of a 64-bit multithreaded PowerPC processor element (PPE) and eight synergistic processor elements (SPEs) connected by a high bandwidth bus in a single chip. The PPE is backed up by two levels of on-chip caches that are coherent with the external main memory. The SPEs are capable of extremely high performance floating-point arithmetic in a SIMD fashion for media and streaming workloads. An SPE has a relatively small private memory (called local store) that is not coherent with the external main memory. This local store is managed by software and explicitly-addressed. Each SPE also has a DMA engine to enable transfer of data between the local store and main memory. All DMA operations have to be explicitly programmed in software, and this significantly affects ease-of-programming. Programming the Cell BE is complicated by its unconventional memory system and by the multiple heterogeneous cores with two different ISAs. Developers of applications for the Cell BE are subject to a programming model that is relatively difficult compared with the model for conventional SMP-like multicores with cache coherence.

In this paper, we propose a runtime system and programming environment called COMIC (COherent shared Memory Interface for Cell BE), which supports a single global address space between the PPE and the SPEs. COMIC provides the programmer with an illusion of a shared memory address space between the PPE and the SPEs, in which each processor can access any data item, without the programmer having to worry about the location of data or how to obtain its value. In our programming environment, the only thing the programmer or the compiler needs to do is to distinguish shared memory reads and writes and replace them with the read and write APIs provided by COMIC. Using the COMIC runtime system and programming environment makes it much easier to program and to port shared memory multithreaded programs to the Cell BE processor while achieving high performance. The major contributions of this paper are the following:

- We describe the design and implementation of the COMIC runtime system and its programming environment. The COMIC programming model supports SPMD-style parallel programming. It employs a new memory consistency model called centralized release consistency (CRC) that supports lazy twin creation with multiple concurrent writers and page-level coherence. Coherence management is centralized in the PPE and no asynchronous messages are sent to an SPE. This allows the SPEs to avoid polling message channels and handling interrupts, and results in better performance.

- We present a new page buffer design for the SPE local store. It is similar to a software cache, but it significantly reduces conflict misses.

- We present the COMIC application programming interface and its use for program optimization. The API can be directly used by the programmer, or used as a target of compilers for shared memory parallel programming languages, such as OpenMP[5] and UPC[21].

- We show the effectiveness of our runtime system, comparing its performance with an alpha version of the IBM XLC OpenMP compiler for the Cell BE [11], using twelve parallel benchmark applications.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 summarizes the Cell BE heterogeneous multicore architecture and method of generating executables. Section 4 presents details of the design and implementation of the COMIC runtime system. Section 5 describes the COMIC API and optimization techniques for COMIC with the API. Section 6 presents the results of evaluating the performance of our COMIC runtime system. Finally, Section 7 concludes the paper.

2. RELATED WORK

There is a lot of previous work done in page-based software shared virtual memory (SVM) for loosely coupled, distributed memory multiprocessors including clusters of workstations or PCs [2, 6, 25, 26, 29, 37, 38, 39, 41, 42]. All these approaches, except for Shasta [36, 37], exploit an MMU's page fault mechanism in each node to detect and handle page faults. Embedding a coherence protocol in the page fault handler of the OS provides coherence at the page level between nodes. Shasta is a software-only approach, similar to COMIC. It implements a shared address space by inserting checks in an executable at every load and store. The inlined code checks if the data is available locally.

There are major differences between COMIC and the previous software SVMs. First, in an SPE, there is no MMU that is specialized for detecting and handling page faults between the local store and main memory.

Second, the local store in an SPE is too small to store coherence information and to cache pages from the shared address space at the same time. Therefore, COMIC stores coherence information in the main memory, and coherence is managed by the PPE. In contrast, previous approaches distribute coherence information and pages between nodes for performance. Making a node the centralized manager is not possible in loosely coupled, distributed memory multiprocessors because the manager node becomes a serious bottleneck [29]. Zhou *et al.*[42] propose Home-based LRC (HLRC). In HLRC, a home location is maintained for each page, and the homes are distributed over multiple nodes. All updates to the page are propagated to the home and all copies are derived from the home. Even though our approach is similar to this approach, our approach has a single home, the PPE.

Another negative effect of the small local store is that protocol messages and page transfers can be very frequent due to local store overflow. This is not the case for previous approaches because overflow is handled by the operating system support for virtual memory and swap space in each node. COMIC handles local store overflow in its coherence protocols.

Finally, polling or interrupt handling for processing incoming, asynchronous messages in the SPE is quite expensive. COMIC uses a coherence protocol in which only an SPE can initiate coherence communication between the PPE and itself. Bilas *et al.*[3] propose a method to avoid asynchronous message processing in SVM systems for SMPs. They extend 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 does not need other resources, such as network processors, to avoid asynchronous messages sent to SPEs. The avoidence mechanism is incorporated into the COMIC coherence protocol.

Previous work has targeted programming the Cell BE using multiple different programming models. The IBM XL compiler uses OpenMP [32]. Fatahalian *et al.*[16] design Sequoia, a language that incorporates the notion of hierarchical memory, and exposes communication and locality in the program. Rabbah [35] and Morita *et al.* [31] propose using a streaming model to program the Cell BE. While COMIC provides a complete programming model by itself, the COMIC runtime API is low-level and can be used to efficiently implement some of these and other programming paradigms. For example, Unified Parallel C [21] is a parallel language that simplifies parallel programming using a globally shared address space. However, there is no UPC compiler available as yet that supports the Cell BE processor. COMIC will be a good target of code generation for a UPC compiler for the Cell BE processor.

There has been some previous compiler work done for the Cell BE. Eichenberger *et al.* [14] present the design and implementation of the IBM XL compilers for the Cell BE. The compiler performs Cell SPU-specific branch prediction, instruction scheduling, data and code partitioning, and SIMD code generation.

O'Brien *et al.* [32] introduce the IBM single-source compiler that supports OpenMP and its relaxed memory consistency model on the Cell BE, and generates different instruction sets for the PPE and SPE from a single-source input. It uses a software cache implementation that guarantees coherence between the local store and main memory. The software cache used in their work is a software implementation of the traditional hardware cache. Its organization is largely different from the COMIC page buffer.

Chen *et al.* [7, 8] present compiler data prefetching techniques for local stores by exploiting loop tiling and static buffers. COMIC does not use any compiler support currently.

Balart *et al.* [1] describe a software caching scheme for the Cell BE that uses a hashed list representation and allows for full associativity. This cache exploits the fully associative feature to increase communication-computation overlap, and thus increases performance. In our work, we provide for large-sized pages together with support for multiple-writers per page, thus exploiting the high bandwidth available on the Cell BE chip to improve performance.

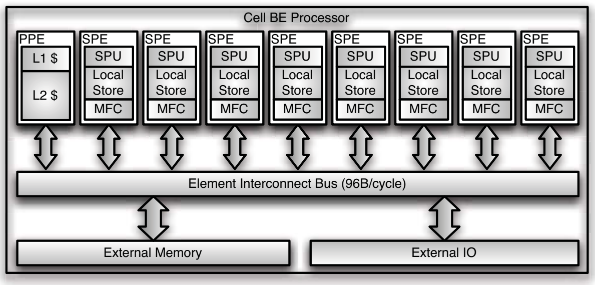


Figure 1: The Cell BE Architecture

3. BACKGROUND

In this section, we briefly review the Cell BE architecture [19, 23] and the process of generating executables.

3.1 Cell BE Architecture

Figure 1 shows the organization of the Cell processor. The PPE is for general purpose processing, such as system management by an OS. It can address resources directly by using an appropriate effective address. The addresses within the main memory are called effective addresses (EAs). The EA is computed when the PPE performs a load, store, branch, or cache instruction, or when it fetches the next instruction. Coherence between the main memory and the caches is guaranteed by the hardware.

An SPE contains a synergistic processor unit (SPU) designed to accelerate media and streaming workloads, a local store, and a memory flow controller (MFC). The SPU supports 128-bit SIMD instructions. The memory instructions access 128 bits of data, and this data must be aligned to 16-byte boundaries. The local store size is 256KB. Addresses within the local store are called local store addresses (LSAs). The MFC includes a DMA engine and provides methods for data transfer and synchronization between main memory and local store, or between an SPE local store and another SPE local store. MFC commands can be issued either by the SPU using channel instructions or by the PPE (or other devices) by performing loads/stores to memory-mapped I/O registers in an MFC.

The primary communication mechanisms between processor elements are DMA transfers, mailbox messages, and signal-notifications that are controlled by the MFC. DMA transfers are for moving data and code between the main memory and a local store. Each DMA command involves both an LSA and an EA. Data is transferred between the local store and the MFC in units of 128 bytes. The MFC supports up to 16 concurrent DMA requests of up to 16 KB each. The MFC guarantees coherence with the L1 and L2 caches of the PPE when data is transferred to/from the main memory. Mailboxes are for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one for receiving messages. The maximum number of outbound mailbox entries is one and that of inbound mailbox entries is four. Signals are for control communication from the PPE or other devices. Signalling uses 32-bit registers that can be configured for one-sender-to-one-receiver or many-senders-to-one-receiver.

COMIC uses mailboxes for control communication and synchronization between the PPE and the SPEs. It uses DMA transfers for data and code movement between main memory and SPE local stores.

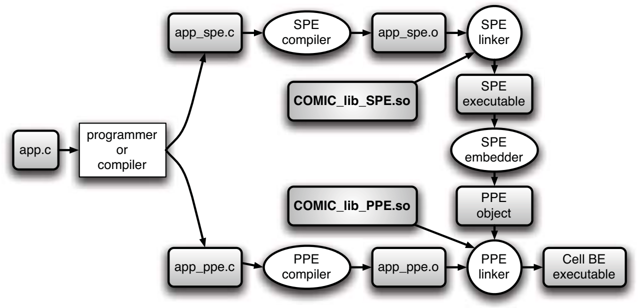


Figure 2: Generating a Cell BE executable.

3.2 Cell BE Binary Generation

Figure 2 shows the common sequence that a programmer typically follows to generate an executable for the Cell BE processor. Given a sequential or an explicitly parallel program, the user or compiler generates two different source files, one for the PPE and the other for SPEs (say `app_ppe.c` and `app_spe.c`).

The source file `app_ppe.c` is the main program of the application, and it is executed on the PPE. The file is compiled with a PPE compiler to generate a PPE object. The file `app_spe.c` is a program that is executed in the SPEs at the request of the main program. It is compiled with an SPE compiler to obtain an SPE object, which is linked together with the SPE libraries to obtain an SPE executable. If the size of the SPE executable is too big for its local store, the functions in the code must be overlaid for execution. The

Cell SDK [22] provides a code-overlay mechanism. To obtain a single binary image that will be loaded into main memory, the PPE embedder embeds the SPE executable into a PPE object. This object is linked together with `app_ppe.o` and PPE libraries to generate a final Cell BE executable image. The COMIC runtime system is provided to the compilers in the form of user-level libraries.

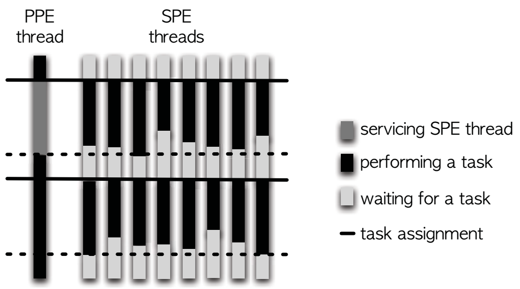


Figure 3: COMIC thread model.

4. THE COMIC RUNTIME SYSTEM

In this section, we describe the design and implementation of the COMIC runtime system.

4.1 Programming Model

A good parallel programming model provides ease of programming while achieving high performance. The COMIC programming model is a shared memory parallel programming model in which threads cooperate in a single shared address space. Writes to a shared location by one thread are visible to reads of other threads. Threads execute on the PPE and the SPEs, one thread per processing element.

The thread running on the PPE is responsible for assigning tasks and providing OS services to SPE threads. The PPE thread also manages synchronization and coherence actions. When SPE threads are performing tasks in which they access the shared address space, the PPE thread cannot perform any computation other than servicing the SPE threads. At other times, the PPE thread is allowed to participate in the main application computation. The PPE thread follows a round-robin policy to service SPE threads to guarantee fairness and to avoid starvation.

Figure 3 illustrates our thread model. The PPE thread creates and terminates SPE threads. When an SPE thread is created, it waits for a task assignment from the PPE thread. After performing the task assigned, it again waits for more tasks assigned from the PPE thread. If there is no need to service the SPE threads, the PPE thread can perform its own task.

4.2 Memory Consistency Model

A well-known memory coherence and consistency protocol for distributed memory multiprocessors is lazy release consistency (LRC) [10, 25, 26]. It is a form of relaxed consistency [10, 17] that alleviates the frequency of coherence messages and data transfers. In a relaxed memory consistency model, coherence actions, such as invalidates or updates due to writes, are postponed until the next synchronization point. Writes become globally visible at the next synchronization point.

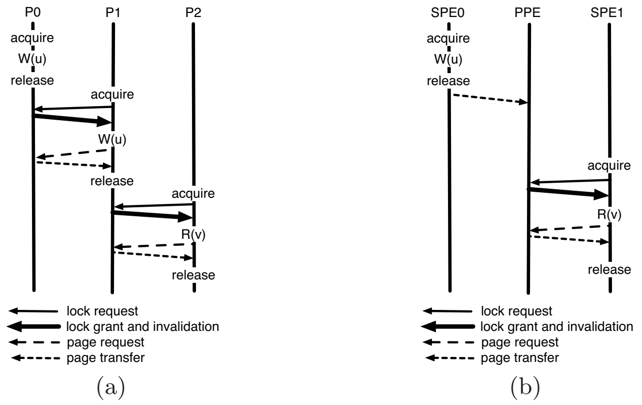


Figure 4: (a) Invalidate-based lazy release consistency. (b) Centralized release consistency.

In release consistency (RC) [17], synchronization operations are divided into two categories: *acquire* and *release*. An acquire is an operation that is performed to gain access to some shared locations (i.e., lock). A release grants permission to another processor to gain access to some shared locations (i.e., unlock). A properly synchronized program on RC produces the same results as it would on the sequentially consistent model [28]. Other synchronization mechanisms can be implemented with acquires and releases. A barrier can be implemented as performing a release then an acquire for all processors that are participating in the barrier.

Relaxed consistency models for the 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 movement between nodes, they may not be appropriate for software SVMs [29]. A software SVM version of RC is the eager release consistency (ERC) [6]. A processor sends invalidate messages to other processors at the release point in invalidate-based ERC. On the other hand, LRC propagates invalidate notices not at the release point, but at the time of an acquire by the other processors. Invalidations are sent to the acquiring processor only for the writes that precede the acquire. LRC aims to reduce the number of coherence messages. Figure 4 (a) shows an example of LRC. At the beginning, all processors have the page p that contains variables `u` and `v`.

Even though LRC is an effective protocol used in previous software SVMs, it is not appropriate for the Cell BE processor. LRC will require the SPEs to receive asynchronous messages during their computation for a task. Processing asynchronous messages will cause the SPEs to digress from performing their task, and will degrade performance. Moreover, the SPE local stores have a limited size and there is no virtual memory implemented in the SPEs. If we adopt LRC, we will be faced with the problem of handling local store overflows.

To avoid these problems, we propose *Centralized Release Consistency* (CRC). The main idea in CRC is that managing locks and pages for coherence is centralized to a single processor, the PPE. No coherence action is initiated by the PPE, and no asynchronous messages are sent to the SPEs. As in LRC, locking occurs and invalidations are sent at acquire points, and locks are relinquished at release points. Also, when a page fault occurs after a coherence invalidation due to an acquire, the faulting SPE fetches the up-to-date version of the faulting page from the main memory through a DMA transfer. However, unlike LRC, modified pages in an SPE local store are sent to main memory when the SPE local store overflows or when the code performs a release. Figure 4 (b) describes CRC. Conceptually, the number of messages sent and received between processors in CRC is the same as that in LRC, except that in CRC a releasing processor updates the shared page in main memory for coherence.

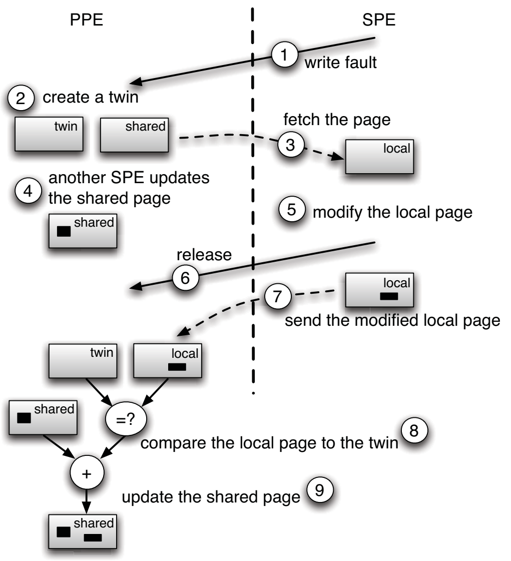


Figure 5: Updating a shared page.

4.3 Multiple-writer Protocol

In CRC, multiple SPEs can write to different locations of the same page at the same time without synchronization. This reduces the number of ping-pong communication between processors due to false sharing. To implement the multiple-writer protocol, the PPE maintains a directory entry for each page. A directory entry contains three fields, read, write, and twins. The read/write fields record the IDs of SPEs that currently have read/write permission to the page. The write field is always a subset of the read field, i.e., whenever an SPE fetches a faulting page when a write fault occurs, the PPE records the SPE ID in both the read and write fields.

For multiple writers, whenever an SPE sees a write fault, the PPE makes a twin (a copy of the page) of the faulting page in main memory. Even though an SPE already has the page for read, it must obtain write permission from the PPE before it first writes to the page. In other words, upgrading the access permission from read to write is treated as a write fault in COMIC. Every writer writes to its own local copy of the shared page. At the next release, the writer sends its modified local page to the PPE. The PPE updates the shared page with the differences resulting from comparing the modified local page with the twin. The shared page's read/write fields in the directory entries are updated. Figure 5 describes the update process.

The PPE maintains an invalidate list for each SPE. The invalidate list of each SPE contains the IDs of pages that need to be invalidated when the SPE performs an acquire operation. Whenever a shared page is released by an SPE and updated in main memory, the page ID is inserted into the invalidate list of each SPE whose ID is listed in the read field of the page's directory entry, except for the updating SPE itself. After an SPE performs an acquire, the pages in the invalidate list for that SPE will be invalidated and the list will be cleared. In addition, when an SPE encounters a write fault for a page, the page ID will be removed from the invalidate list for that SPE if the ID is in the list.

4.4 Lazy Twin Creation

Since creating a twin is an expensive operation (i.e., main memory allocation), COMIC creates twins lazily. Instead of creating a twin whenever a write fault occurs, the PPE postpones the twin creation of a page until a dirty copy updates the page in the main memory. To do so, each page in the main memory has a pointer to its twin (twin pointer) for each SPE. Just before a page update (i.e., write-back) by an SPE occurs in the main memory, the PPE creates a twin for the page. Each time a twin is created, the set of SPE twin pointers that get updated are the write set in the directory entry for the page, minus the set of SPE twin pointers pointing to previously created twins. The PPE knows which SPEs currently have write permission by looking at the write field of the page's directory entry. Our lazy twin creation is an application of the copy-on-write mechanism found in virtual memory operating systems.

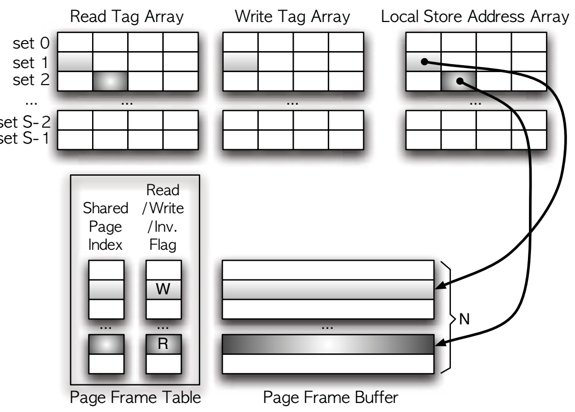
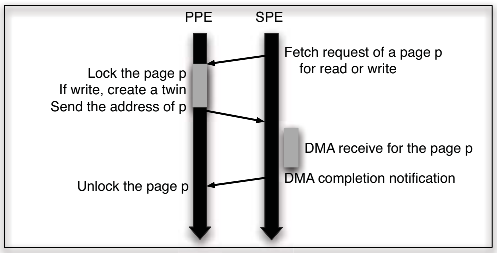


Figure 6: The organization of COMIC software cache in local store.

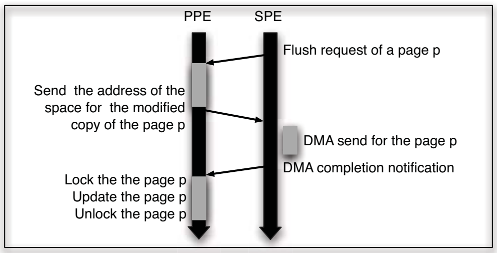
4.5 Page Buffer in Local Store

The COMIC runtime system includes a page buffer implemented in local store. The page buffer is a software cache that consists of two tag arrays (one each for read and write), a local store address array, a page frame table, and page frame buffer. Figure 6 shows its organization with S sets and N page frame buffers. It is similar to a 4-way set associative cache, but it has two tag arrays, one for reads and the other for writes. Each way in a set is associated with an element of the read tag array, an element of the write tag array, and an element of the local store address array. The associated elements are accessed with the same array index. The read and write tag arrays store the global addresses of pages. The local store address array stores the local addresses of pages in the page frame buffer.

For a given shared memory access with an address A, the hashing mechanism to find a set index for A is similar to that of the hardware 4-way set associative cache. For tag


PPE
SPE
Fetch request of a page p for read or write
Lock the page p
If write, create a twin
Send the address of p
DMA receive for the page p
DMA completion notification
Unlock the page p

(a)


PPE
SPE
Flush request of a page p
Send the address of the space for the modified copy of the page p
DMA send for the page p
DMA completion notification
Lock the the page p
Update the page p
Unlock the page p

(b)

Figure 7: Three-way handshaking between the PPE and an SPE. (a) Fetching a page from main memory. (b) Writing back a modified local page to main memory.

matching, the global address P for A is computed. If the access is a read, the four elements of the read tag array in the set are compared to P. The 128-bit SIMD instructions are exploited for this comparison. If there is a tag match, the corresponding element in the local store address array contains the local address of the frame buffer that corresponds to A. The page offset of A is added to the frame buffer address to obtain the local address. If there is no matching tag, then a read page fault occurs and the COMIC page fault handler is invoked. The handler is in charge of the three-way handshaking explained later for fetching the page.

If the access is a write, the four elements of the write tag array in the set are searched first. If there is no matching element, the four elements of the read tag array are compared. If there is no matching element, a write page fault occurs. The fault handler requests the faulting page for writes from the PPE. Otherwise, since it is a read hit, the SPE only needs to obtain write permission from the PPE.

When a faulting page is fetched from main memory and the page frame buffer is full, a page in the buffer is flushed to main memory to make space for the page fetched. The pages in the page frame buffer are replaced in a round-robin manner. The Read/Write/Invalid flag in the page frame table is appropriately set according to the type of access obtained for the page. The page index for a page in the page frame table is the address of the page in main memory. Modified pages that overflow from an SPE local store are treated in the same way as those flushed to main memory at a release point.

The COMIC API provides functions for reading and writing shared variables through the software cache. In other words, all shared variable accesses in the SPE code need to be replaced by a read or write function in the API. the API functions have the formats below, where `TYPE` is one of `char`, `short`, `int`, `long`, `long long`, `float`, or `double`.

```
TYPE COMIC_spe_read_TYPE(TYPE *ptr);
void COMIC_spe_write_TYPE(TYPE *ptr, TYPE v);
```

The major difference between the COMIC software cache and conventional hardware caches[20] or existing software caches [14, 30] is the number of frames (i.e., cache lines) in the page frame buffer and the separation of read and write tag arrays. In the COMIC software cache, the number of sets multiplied by the number of ways is greater than the number of page frames in the buffer ($4 \times S > N$ in Figure 6) in order to reduce conflict misses. Separated read and write tags remove an extra check for the read/write flag after a cache hit when the access is a write, resulting in performance improvement.

4.6 Avoiding Asynchronous Messages Sent to SPEs

We want to avoid sending asynchronous messages to the SPEs because they are detrimental to SPE performance since they either require the SPEs to regularly poll the message channels, or require invoking an SPE interrupt handler to handle the messages. In the COMIC runtime system, all coherence actions are initiated by SPEs using three-way handshakes. Figure 7 describes different three-way handshakes employed between the PPE and an SPE: fetching a page from main memory (Figure 7(a)) and writing back a modified local page to main memory (Figure 7(b)).

After an SPE initiates a DMA transfer, it updates the page buffer bookkeeping information in parallel with the DMA transfer. Due to the limited size of the local store, the case of flushing a modified local page and then fetching a new page frequently occurs. COMIC combines these two contiguous three-way handshakes into a single three-way handshaking protocol for this specific to reduce message traffic.

Sending and receiving messages in the three-way handshakes are implemented using mailboxes. Since there is only one entry for the outbound mailbox of an SPE, sending DMA completion notification from an SPE to the PPE may block the next request from the SPE if the message is not serviced by the PPE on time. Therefore, we use a 128B DMA transfer to implement the DMA completion notification message to the PPE. The PPE polls a specific location for each SPE in the main memory to check whether the notification has arrived.

4.7 Locks and Barriers

The PPE maintains locks in a centralized manner. Similar to other requests, the requests for locks are serviced in a round-robin manner by the PPE. To acquire a lock, the requesting SPE sends a message through the mailbox to the PPE. If the lock is available, the PPE sends a grant message to the requester and records the SPE ID as the owner of the lock. Otherwise, the requester is blocked, and the PPE records the requester ID in the list of SPEs blocked waiting for the lock. When the lock is released, the releaser informs the PPE through the mailbox. The PPE grants the lock to the next SPE whose ID is in the list of SPEs blocked waiting for the lock.

COMIC supports a centralized barrier implementation. The PPE maintains a counter for each barrier. The counter records the number of SPEs that have arrived at the barrier. Whenever an SPE arrives at the barrier, it sends a message to the PPE to increment the counter for the barrier. If the counter equals the number of SPEs that is defined by COMIC, the PPE sends a message through the mailbox to each SPE waiting on the barrier.

```
#include <stdio.h>
#include <omp.h>
#define SIZE 24

int a[ SIZE ];

int main() {

  int i;

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

  for( i=0; i < SIZE; i++ )
    printf( "%d\n", a[ i ] );

  return 0;
}
```

(a)

```
#include <stdio.h>
#include <comic_ppe.h>

#define SIZE 24

COMIC_shared (
  int a[ SIZE ];
)

int main() {

  int i;

  COMIC_ppe_init();
  COMIC_ppe_run( 0 );

  for( i=0; i < SIZE; i++ )
    printf( "%d\n",
      COMIC_access( a[ i ] ) );

  COMIC_ppe_exit();

  return 0;
}
```

(b)

```
#include <comic_spe.h>
#define SIZE 24

COMIC_shared (
  int a[ SIZE ];
)

void spe_main_case0() {

  int i, tid, chunk, low, high;

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

  for( i = low; i < high; i++ )
    COMIC_spe_write_int( &COMIC_access(a[i]), i );

  COMIC_spe_barrier( 0 );
}

COMIC_spe_main (
  case 0:
    spe_main_case0();
    break;
)
```

(c)

Figure 8: (a) An OpenMP program. (b) COMIC program for the PPE thread. (c) COMIC program for SPE threads.

5. COMIC API AND OPTIMIZATIONS

In this section, we describe the COMPIC API functions and optimization techniques for them.

5.1 API Functions

The COMIC API provides functions for reading and writing shared variables, thread creation, destruction, page size variation, synchronization, and COMIC specific optimizations.

Figure 8 gives an example of how to write a COMIC parallel program with the APIs. For simplicity, we explain how an explicitly parallel program can be converted into a COMIC parallel program. Figure 8 (a) is an OpenMP parallel program written in C. The first `for` loop is a parallel loop, and multiple threads can execute different iterations of the loop in parallel. It is converted to the two C programs with COMIC APIs in Figure 8 (b) and (c).

Figure 8 (b) is the PPE program that becomes the PPE thread. Array `a` is declared as a global shared array. The PPE thread creates SPE threads (`COMIC_ppe_init();`) and assigns a task (the task ID is equal to 0) to each SPE thread (`COMIC_ppe_run( 0 );`). Each thread executes the same SPMD-style code fragment that corresponds to the task ID.

Figure 8 (c) is the SPMD-style SPE program that each SPE thread executes. The main function in the SPE program loops until the PPE thread assigns a task. After a task is assigned, the case statement that matches the task ID is executed. After finishing the task, the SPE thread loops again. The statements in `case 0` are the code that each SPE executes for task 0. Each thread obtains a chunk of the iterations of the `for` 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.

```
{
  int *a_page_bound = 0;
  int *a_local;
  for(i=low; i<high; i++) {
    if(&a[i] > a_page_bound) {
      /* compute the EA of a[i]'s current page bound */
      a_page_bound = ...
      /* get the local address of a[i] */
      a_local = COMIC_spe_write_LSA(&a[i]);
    } else {
      a_local++;
    }
    *a_local = i;
  }
  COMIC_spe_barrier( 0 );
}
```

(a)

```
COMIC_round_robin_buffer_begin(4096);
{
  int *a_page_bound = 0;
  int *a_local;
  for(i=low; i<high; i++)
    COMIC_spe_write_local_int(&a[i],i,a_page_bound,a_local);
  COMIC_spe_barrier( 0 );
}
COMIC_round_robin_buffer_end();
```

(b)

Figure 9: Access localization. (a) After converting the `for` loop in Figure 8(c) for access localization. (b) Access localization using COMIC API functions.

5.2 Optimizations for COMIC Runtime

Variable page size. The optimal page size for the COMIC software cache varies for different parallel regions depending on their access patterns. For example, a bigger page size typically will be better for a parallel loop with sequential access patterns, and a smaller page size typically will be better for a loop with random access patterns, even though it is not easy to apply a general rule. For this purpose, COMIC provides the API functions below to specify a different page size for each parallel region. Whenever the page size is changed, the SPE flushes and invalidates the pages in the software cache.

```
void COMIC_page_size(int page_size);
```

Access localization. If a shared array is accessed in a parallel loop by an SPE with a stride that is less than the page size, the first access in a page for the array experiences a page fault. After the page is fetched from the main memory to the software cache, the remaining accesses of the array in the same page will be to the SPE local store if the page is not kicked out from the software cache due to conflict or capacity misses. For example, the `for` loop in Figure 8(c) is converted to the code in Figure 9(a) to localize the array access `a[i]`. The API function `COMIC_spe_write_LSA(&a[i])` returns the local store address (LSA) of `a[i]` if it is a hit in the cache. Otherwise, it fetches the page for `a[i]` and returns the LSA for `a[i]`.

If all shared accesses in a parallel loop are strided array accesses, and the number of statically different access instances is less than the number of page frames in the software cache, the software cache can be managed differently to localize these shared accesses based on the above observation. For such parallel loops, the software cache does not use the indexing mechanism described in Section 4.5. Instead, the page frame buffer is directly accessed and managed in a round-robin manner to cache fetched pages for the accesses. COMIC provides the API functions below for this special case.

```
void COMIC_round_robin_buffer_begin(int page_size);
void COMIC_round_robin_buffer_end();
TYPE COMIC_spe_read_local_TYPE(TYPE *ptr,
                TYPE *page_bound, void *local_lsa);
void COMIC_spe_write_local_TYPE(TYPE *ptr, TYPE v,
                TYPE *page_bound, void *local_lsa);
```

For example, the `for` loop in Figure 8 is converted into the code in Figure 9 (b) using these API functions. This code is equivalent to the code in Figure 9 (a).

Read only accesses. If we know that a page is only read (not written) by the SPEs in a parallel region, the three-way handshaking for fetching is not necessary between the PPE and SPEs in the parallel region. In this case, the reading SPE obtains the page directly from the main memory through a DMA transfer without sending/receiving any messages to/from the PPE. COMIC provides the following API function for those read-only accesses.

```
TYPE COMIC_spe_read_only_TYPE(TYPE *ptr);
```

Single writers. Moreover, if we know that a page is written by only one SPE in the parallel region (i.e., there is no false sharing in the parallel region), the three-way handshaking for both fetching and flushing this page are not necessary. We just use a DMA transfer for fetching/flushing this page from/to the main memory in the parallel region. To enable this optimization for a parallel loop, false sharing elimination and loop restructuring techniques similar to those described in [13, 24] are needed. COMIC provides the following API function for the case of single writers.

```
void COMIC_spe_write_single_TYPE(TYPE *ptr, TYPE v);
```

Assigning tasks to the PPE. If we can guarantee that a parallel region has no messages sent or received, we can assign some task to the PPE because it does not need to service SPEs in this parallel region. Typically, such parallel regions are exposed after applying the above two optimizations for read-only accesses and single writers.

Table 1: Applications Used

App.	Source	Input	Shared Data	Page Buffer	# of Seg*
ammp	SPEC	reference	3.9MB + malloc	128KB	5
CG	NAS	class A	26.1MB	192KB	3
EP	NAS	class A	16.2KB	192KB	1
equake	SPEC	reference	28.5MB	192KB	1
FT	NAS	class A	417.7MB	168KB	3
IS	NAS	class A	64MB + malloc	192KB	1
jacobi	OpenMP	2000 × 2000	91.6MB	192KB	1
md	OpenMP	8192 particles	768KB	192KB	1
MG	NAS	class A	464B + malloc	176KB	10
SP	NAS	class A	79.6MB	192KB	10
STREAM	HPCC	N=1, Ns=1000	1.9MB	192KB	1
SWIM	SPEC	reference	190.6MB	192KB	6

*The number of segments in code overlays.

Table 2: System Configurations

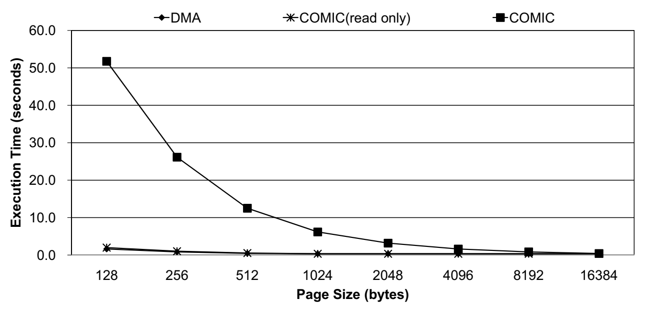
Cell Blade Server	
Configuration	Dual 3.2GHz Cell BE, 8 SPEs each 512KB L2 cache, 2GB main memory RedHat Linux ES 5.1
OpenMP	IBM XL Cell cbexlc, -O5
COMIC	IBM XL Cell ppuxlc, -O5 IBM XL Cell spuxlc, -O5 IBM XL Cell ppu32-embedspu (embedder)
Intel Xeon Server	
Configuration	1.6GHz Xeon, Dual socket (Clovertown, quadcore, E5310) total 8 cores A shared 4MB L2 $ for each pair of cores 12GB main memory SUSE Linux ES 10.0
OpenMP	GNU gcc 4.2.3, -O2 Intel icc 10.1, -O2

6. PERFORMANCE EVALUATION

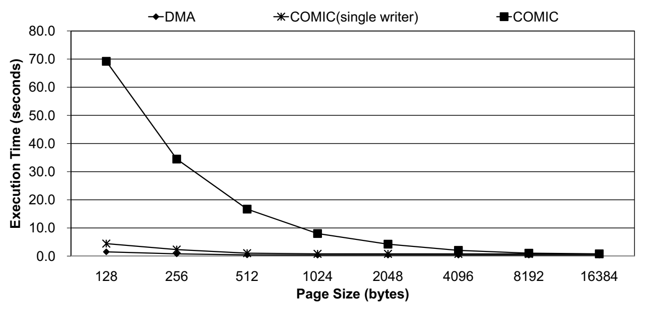
This section presents the evaluation environment and performance results for the current COMIC implementation.

6.1 Benchmark Applications

To evaluate the performance of our runtime system, we use twelve shared memory applications written in OpenMP. They are summarized in Table 1 and from various sources, NAS[12], SPEC OMP [9], HPC Challenge[40], and OpenMP samples in [4]. Since some benchmark applications (CG, EP, FT, IS, MG, SP, and SWIM) are written in FORTRAN, we use the C versions obtained from the Omni OpenMP Compiler Project [34], or obtained by manual translation (SWIM) to a C program. The input data set for each application is shown in Table 1.



(a)



(b)

Figure 10: The cost of DMA transfer and the three-way handshake protocols. (a) For fetching pages. (b) For flushing pages.

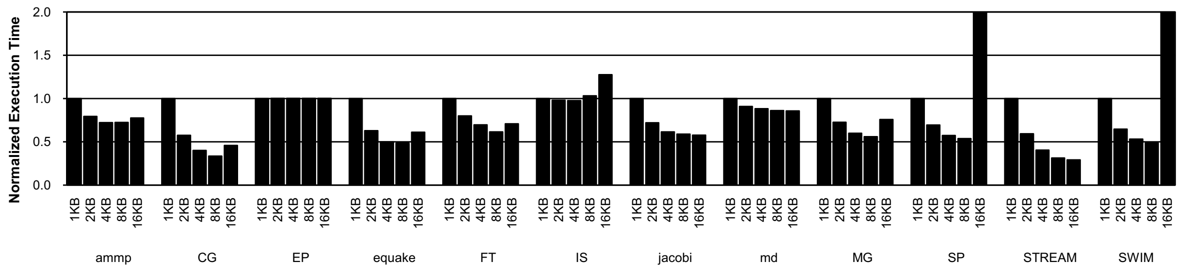


Figure 11: Normalized execution time with 8 SPEs for varying page sizes.

To generate parallel programs that use the COMIC runtime system, we convert the applications manually to programs using COMIC APIs. We faithfully follow the OpenMP annotations of the original OpenMP programs in our manual translation. Table 1 also shows the amount of shared data and the size of page buffers used in each application written in COMIC. The last column shows the number of segments in the code overlays of each application. Due to the small local store, a large application needs to be divided into overlayed code sections to run on an SPE.

The COMIC runtime system is implemented using the Cell SDK library [22]. We evaluate the performance of our COMIC runtime system using an IBM Cell blade server. Table 2 summarizes the parameters of the target machines and compilers used in our experiments. We only use one Cell BE processor in the server. The single-source compiler (cbexlc) supports the OpenMP relaxed memory consistency model on the Cell BE, and generates different instruction sets for the PPE and the SPEs from a single-source input. It uses a software cache implementation that guarantees coherence between the local stores and main memory [14, 22, 32]. However, it does not support any option with which the programmer controls the cache line size at runtime.

6.2 Basic Operation Costs

Figure 10 (a) and (b) show the costs (execution time) of our three-way handshake protocols (COMIC) and DMA transfers (DMA) for fetching and flushing pages, respectively. To obtain these numbers, we program each SPE to access a different 40MB main memory area with a sequential access pattern. All 8 SPEs repeat this process simultaneously until the amount of memory accessed equals 1GB for each SPE. (read only) means that the pages are read-only. In this case, the COMIC three-way handshake for fetching becomes just a DMA transfer with some function call overhead, without any messages being sent or received. (single writer) denotes the case where each SPE writes a different 40MB main memory area. In this case, the flushing protocol becomes just a DMA transfer with some function call overhead. The page size varies from 128B to 16KB. The smaller the page, the more frequent is the transfer.

For fetching pages (Figure 10 (a)), the coherence protocol processing cost is much higher than the cost of a direct DMA transfer. Smaller page sizes manifest this fact due to frequent transfers. However, the protocol optimization for read-only accesses significantly reduces the protocol overhead, to the point of making the overhead almost irrelevant. The protocol overhead is amortized by the transfer time for larger page sizes.

Flushing pages (Figure 10 (b)) is similar to fetching pages. The coherence protocol processing cost is much higher than the cost of a direct DMA transfer.

The time taken to acquire a lock and to release the lock soon after acquiring it is 6.47 μ seconds. The time to perform a barrier is 3.14 μseconds. The reason why an acquire and a release take more time than a barrier is that more messages are involved in a lock than a barrier if all 8 SPEs contend for the lock.

6.3 Sensitivity to the Page Size

We measure COMIC's sensitivity to various page sizes in Figure 11. We vary the page size from 1KB to 16KB and run the applications with 8 SPEs. 16KB is the maximum amount of data that can be transferred through a single DMA transfer in the Cell BE processor. All the numbers are normalized to the time for 1KB page size for each application. When we perform the experiment, the number of sets (S) is set to a 2's power K that satisfies $4 \times K > N$ (refer to Section 4.5). All applications except FT show zero conflict misses in the page buffer with this setup.

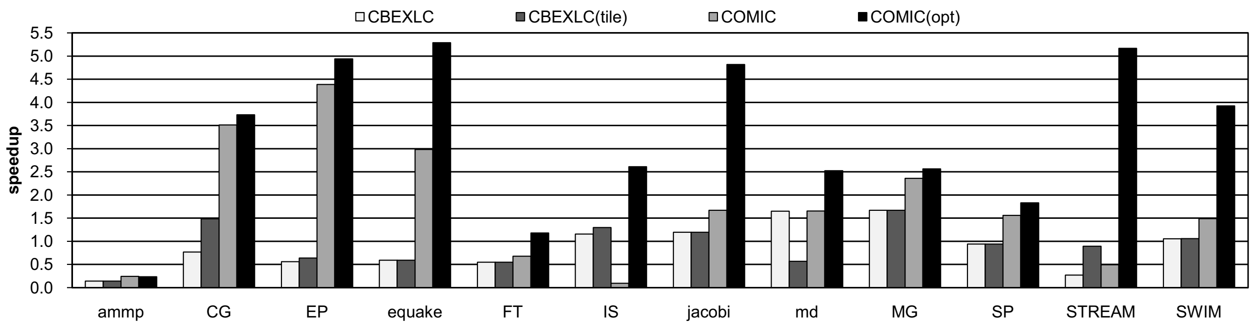


Figure 12: Performance comparison with XL C/C++ Alpha Edition for Multicore Acceleration single-source compiler from IBM alphaWorks. The speedup is obtained over a single PPE.

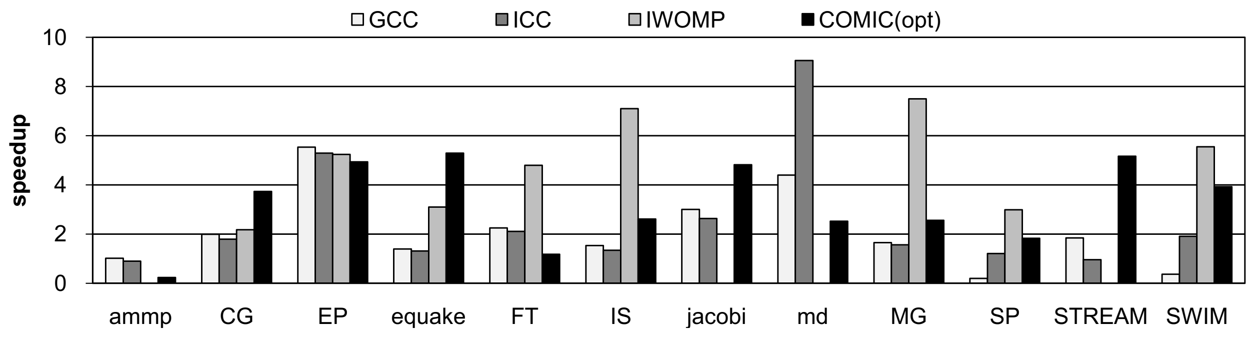


Figure 13: Performance comparison of Cell BE with COMIC runtime and a Xeon quadcore two socket (8 cores) with the gcc OpenMP compiler and Intel C++ OpenMP compiler.

The optimal page size varies for most applications, and it is difficult to pinpoint one page size that fits all. 8 KB is optimal for 6 of 12 applications. jacobi, md, and STREAM perform best with 16KB pages. This implies that they have good spatial locality in a page. The case of 16KB in SP and SWIM causes thrashing in the COMIC page buffer. The working set of some of their parallel loops does not fit in the page buffer with 16KB pages. EP has a very small number of shared memory accesses (in the order of hundreds). IS includes subscript of subscript references. Its access pattern is irregular and spread across pages even if the page size is large (16KB). CG and FT also have irregular access patterns, but for CG, the distance between accesses resides in a page, up to a page size of 8KB.

6.4 Execution Time

Figure 12 shows the result of comparing the speedup of COMIC with that of an alpha version of the single-source OpenMP compiler included in the Cell SDK 3.0 from alphaWorks[11]. The speedup is obtained when using 8 SPEs over 1 PPE. The single-source compiler generates code that exploits the software cache described in [32]. The software cache guarantees coherence between the local stores and main memory, and its line size is 128B. The single-source compiler performs an optimization that is specific to DMA transfers, called DMA tiling, that exploits loop tiling techniques and static buffering as described in [7]. The bars labeled CBEXLC show the speedup of the single-source OpenMP compiler without DMA tiling optimization. CBEXLC(tile) labels the speedup obtained with DMA tiling. The COMIC label denotes the speedup obtained using COMIC without the optimizations described in Section 5.2. The best page size obtained in the prior sensitivity analysis is used for each application. COMIC(opt) labels the speedup obtained using COMIC with the optimization techniques described in Section 5.2, including the variable page size, access localization, optimization for read-only accesses and single writers, and task assignment to the PPE.

COMIC performs as well as or better than CBEXLC for all but IS. Ammp has critical sections protected by locks. Frequent contention for locks accounts for the slowdown. Locks in ammp cause many coherence messages per shared memory access. As mentioned before, IS has irregular access patterns. Many page faults due to its subscript of subscript references account for its slowdown.

Table 3: Optimizations Performed

App.	Access localization	Variable page size	Read-only accesses	Single writer	Tasks to the PPE
ammp	X		X		
CG	X		X	X	
EP	X		X	X	X
equake	X		X	X	
FT	X	X	X	X	
IS	X		X	X	X
jacobi	X		X	X	X
md	X		X	X	X
MG	X		X	X	
SP	X		X	X	
STREAM	X		X	X	X
SWIM	X		X	X	

Moreover, COMIC(opt) performs much better than CBEXLC (tile) for all applications. The optimization techniques described in Section 5.2 are not effective for ammp. Locking overhead elimination is not the goal of those optimizations.

FT and SP show a relatively small speedup compared to others. The reason for FT is that its irregular access pattern causes many conflict misses in the page buffer. SP has a relatively high read fault rate and a large number of coherence messages per access caused by the barriers of many parallel loops.

The optimization techniques for read-only accesses and single writers completely eliminate coherence protocols between the PPE and SPEs for EP, IS, jacobi, md, and STREAM, resulting in enabling task assignment to the PPE. We summarize those effective optimization techniques for each application in Table 3.

6.5 Comparison to a Homogeneous Multicore

To show that the scalability of the Cell BE with COMIC runtime is competitive with the performance achieved using a homogeneous multicore, we show the speedup obtained by running our benchmark applications compiled with the gcc OpenMP compiler (labeled GCC) and Intel C++ OpenMP compiler (labeled ICC) on an Intel Xeon quadcore two-socket (total 8 core) server in Figure 13. In addition, since the numbers obtained from the alphaWorks compiler version are not as good as the numbers reported in [32, 33], we compare with the best available prior results from [32, 33] at this time (labeled IWOMP for CG, EP, equake, FT, IS, MG, SP, and SWIM). COMIC(opt) performs better than IWOMP in two cases (CG and equake), performs almost as well for EP, but performs worse in the 4 remaining cases (FT, IS, MG, and SP). However, note that compiler prefetching optimizations that boost performance in the IWOMP results are complementary to the approach adopted by COMIC, and can be performed by a compiler targeting the COMIC runtime as its programming interface.

The scalability of the Cell BE with COMIC runtime is as good as or better than the homogeneous multicore with the gcc OpenMP compiler for all applications except ammp, EP, FT, and md. The comparison of COMIC with the case of the Intel C++ OpenMP compiler is similar to the case of gcc.

Even though it is not an apples-to-apples comparison, to directly compare the performance of the Cell BE with COMIC runtime to that of the homogeneous multicore with gcc OpenMP (GCC) and icc OpenMP (ICC), we compute the cost/performance ratio. GCC/ICC is, on average, 1.55/1.89 times faster than COMIC(opt) when we directly compare their parallel execution times. A Cell BE chip contains 234 million transistors and a Xeon quad core chip contains 583 million transistors, and two Xeon chips (8 cores) contain 1166 million transistors. This means that COMIC(opt) is 4.98 times cheaper than GCC/ICC. This implies that COMIC(opt) achieves 3.21/2.63 times better performance than GCC/ICC with the same transistor cost.

6.6 Ease of Programming

The COMIC runtime is a low-level shared memory API. Since coherence and consistency is provided by COMIC, programmers do not need to worry about them. In no way does COMIC further complicate programming over the base Cell BE programming model. At least, programming with COMIC is easier than programming with the SDK provided by the vendor, and COMIC achieves high performance.

While this has not been validated with the experience of others, the most difficult task in the manual porting of OpenMP programs is identifying all shared memory accesses. For OpenMP programs with clearly demarcated parallel regions, it is relatively easy to identify shared accesses, and also to determine within the specific parallel region if certain accesses are streaming, read-only or write-once. Both shared access identification and code transformations (including optimizations) can be automated.

The applications were ported to the COMIC runtime by first year graduate students who do not have any parallel programming background. They started to learn OpenMP semantics and finished porting the OpenMP applications to the COMIC runtime in approximately one month.

7. CONCLUSIONS

In this paper, we presented the design and implementation of the COMIC runtime system and its programming environment for the Cell BE architecture. We aimed to develop a flexible runtime system that delivers high performance and is easier to program with compared to using the API provided by the raw Cell BE system.

The COMIC runtime system allows threads to assume a globally shared memory even though they execute on different SPEs with their own private local stores. To support a single shared address space, we proposed a new memory consistency model, centralized release consistency, which is a variation of lazy release consistency with page-level coherence. Centralized management of synchronization and coherence actions makes SPEs avoid receiving asynchronous messages, resulting in no polling and interrupt handling at the SPE side. We presented COMIC's multiple-writer protocol with lazy twin creation. We also proposed a new software cache (page buffer) architecture that significantly reduces conflict misses in the page buffer.

Evaluation results show that COMIC runtime is effective and promising. COMIC provides more flexibility and ample opportunities for optimizations compared to OpenMP, while still abstracting away the complexity of the Cell architecture. Its cost/performance trade-off is better than SMP-like homogeneous multicores. It is a lower level API that enables programmers to selectively hand-tune parts of their Cell applications. COMIC can also be the target of compilers for higher-level programming models (UPC, OpenMP, and the like).

8. REFERENCES

[1] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O'brien, and Kathryn O'Brien. A novel asynchronous software cache implementation for the cell/be processor. In *LCPC '07: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[2] Brian N. Bershad and Matthew 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] Angelos Bilas, Cheng Liao, and Jaswinder Pal 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.

[4] OpenMP Architecture Review Board. OpenMP. `http://www.openmp.org`.

[5] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, version 2.5 edition, May 2005.

[6] John B. Carter, John K. Bennett, and Willy 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] Tong Chen, Zehra Sura, Kathryn M. O'Brien, and John K. O'Brien. Optimizing the use of static buffers for dma on a cell chip. In *LCPC '06: Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing*, pages 314–329, November 2006. Also in Lecture Notes in Computer Science 4382, Springer 2007.

[8] Tong Chen, Tao Zhang, Zehra Sura, Kathryn O'Brien, Kevin O'Brien, and Marc Gonzalez Tallada. Prefetching irregular references for software cache on cell. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[9] Standard Performance Evaluation Corporation. SPEC 2000. `http://www.spec.org/benchmarks.html`.

[10] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[11] IBM DevloperWorks. Cell broadband engine resouce center. `http://www.ibm.com/developerworks/power/cell/downloads.html`.

[12] NASA Advanced Supercomputing Division. NAS parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[13] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. In *ICPP '91: Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 377–381, August 1991.

[14] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 4th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, September 2005.

[15] B. Flachs et. al. A Streaming Processing Unit for a CELL Processor. *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2005.

[16] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 Supercomputing Conference*, November 2006.

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John 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.

[18] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 1–8, May 2006.

[19] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(02):10–24, March/April 2006.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture*. Morgan Kaufmann, fourth edition, 2006.

[21] Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*, pages 63–73, June 2003.

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

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

[24] Tor E. Jeremiassen and Susan J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, New York, NY, USA, July 1995. ACM.

[25] Peter Keleher, Alan L. Cox, and Willy 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.

[26] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 115–132, January 1994.

[27] M. Kistler, M. Perrone, and F. Petrini. CELL Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3), May/June 2006.

[28] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[29] Kai Li and Paul 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.

[30] Jason E. Miller and Anant Agarwal. Software-based instruction caching for embedded processors. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–302, October 2006.

[31] M. Morita, T. Machino, M. Guo, and G. Wang. Design and implementation of stream processing system and library for CELL broadband engine processors. In *Proceedings of the 2007 Parallel and Distributed Computing and Systems Conference*, November 2007.

[32] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. In *IWOMP '07: Proceedings of the International Workshop on OpenMP*, June 2007.

[33] Kevin O'Brien, Kathryn M. O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[34] Parallel and High Performance Applicational Software Exchange Editorial Committee. Omni OpenMP compiler project. `http://phase.hpcc.jp/omni`.

[35] Rodric Rabbah. Beyond gaming: Programming the PLAYSTATION3 Cell architecture for cost-effective parallel processing. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis*, 2007.

[36] Daniel J. Scales, Kourosh Gharachorloo, and Anshu 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.

[37] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan 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.

[38] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David 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.

[39] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael 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.

[40] HPC Challenge Team. HPC challenge benchmark. `http://icl.cs.utk.edu/hpcc/`.

[41] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian 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.

[42] Yuanyuan Zhou, Liviu Iftode, and Kai 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.