Cell GC: Using the Cell Synergistic Processor as a Garbage Collection Coprocessor

Chen-Yong Cher Michael Gschwind
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
chenyong@us.ibm.com mkg@us.ibm.com

Abstract

In recent years, scaling of single-core superscalar processor performance has slowed due to complexity and power considerations. To improve program performance, designs are increasingly adopting chip multiprocessing with homogeneous or heterogeneous CMPs. By trading off features from a modern aggressive superscalar core, CMPs often offer better scaling characteristics in terms of aggregate performance, complexity and power, but often require additional software investment to rewrite, retune or recompile programs to take advantage of the new designs. The Cell Broadband Engine is a modern example of a heterogeneous CMP with coprocessors (accelerators) which can be found in supercomputers (Roadrunner), blade servers (IBM QS20/21), and video game consoles (SCEI PS3). A Cell BE processor has a host Power RISC processor (PPE) and eight Synergistic Processor Elements (SPE), each consisting of a Synergistic Processor Unit (SPU) and Memory Flow Controller (MFC).

In this work, we explore the idea of offloading Automatic Dynamic Garbage Collection (GC) from the host processor onto accelerator processors using the coprocessor paradigm. Offloading part or all of GC to a coprocessor offers potential performance benefits, because while the coprocessor is running GC, the host processor can continue running other independent, more general computations.

We implement BDW garbage collection on a Cell system and offload the mark phase to the SPE co-processor. We show mark phase execution on the SPE accelerator to be competitive with execution on a full fledged PPE processor. We also explore object-based and block-based caching strategies for explicitly managed memory hierarchies, and explore to effectiveness of several prefetching schemes in the context of garbage collection. Finally, we implement Capitulative Loads using the DMA by extending software caches and quantify its performance impact on the coprocessor.

Categories and Subject Descriptors C.3 [*Computer Systems Organization*]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS; D.2.11 [*Software*]: SOFTWARE ENGINEERING—Software Architectures; D.3.3 [*Software*]: PROGRAMMING LANGUAGES—Language Constructs and Features

General Terms Algorithms, Performance

Keywords Cell, SPU, SPE, BDW, garbage collection, mark-sweep, coprocessor, accelerator, local store, explicitly managed memory hierarchies

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'08 March 5–7, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-796-4/08/03. . . $5.00

1. Introduction

Traditional computer systems use caches to reduce average memory latency by providing non-architected temporary high-speed storage close to microprocessors. Alas, while this design choice reduces average latency and presents the programmer with a "flat" memory hierarchy model, the cost of maintaining this illusion is significant. This cost includes area to store cache directories, perform tag match, and implement cache miss recovery; impact of cache miss logic on cycle time or complex speculative logic; and runtime costs like coherence traffic which is increasing as the number of processors are scaled up in multiprocessor systems.

To counter this trend and offer low latency storage with guaranteed access time, an increasing number of designs are offering fast, architected on-chip storage. Among these designs is the Cell Broadband Engine [16, 11, 5, 12], which offers eight high-performance RISC-based processor cores with dedicated architected low latency storage in the form of the SPE local store.

Architected local memories are most commonly used to store a set of processor-local data, or contain working copies of large data sets. While local storage offers significant potential for processing with data-intensive applications [12, 20, 9], little exploration has been performed into the use of local memories for managed runtime-environments, and automatic memory management with garbage collection, which are increasingly adopted to simplify program development and maintenance.

Noll *et al.* [18] explore the use of Cell SPE cores for executing Java applications. Alas, many system functions, such as type resolution, and garbage collection are performed on the central PPE which may threaten to become a bottleneck if significant speedups are accomplished with SPE-based execution.

In this work, we explore the idea of a Garbage Collector Coprocessor by running Java applications with the Boehm-Demers-Weiser (BDW) mark-sweep garbage collector on a live Cell Processor. In this work, we concentrate on optimizing GC performance for a processor with an explicitly managed local store by managing the local store using caching and prefetching techniques with the DMA copy engine.

In porting the garbage collector to the Cell BE, our emphasis was the exploitation of local stores with copy-in/copy-out semantics. Exploiting local memories with explicit copy semantics and block copy facilities is applicable to a growing class of embedded processor designs.

To explore the techniques and performance attributes of the explicitly managed memory hierarchy, we focus on this attribute of the Cell BE architecture exclusively. The Cell BE architecture offers additional capabilities such as multicore thread level parallelism and integrated SIMD capabilities which are not explored within the scope of this work.

Results presented here were measured on an experimental Cell BE blade with 2 Cell BE chips for a Cell BE system configuration with 2 PPEs and 16 SPEs executing Cell BE Linux 2.6.20-CBE. The operating frequency of the system was 3.2GHz, and the system was

populated with 1GB of main memory. The host processor runs the Java application and offloads its marking work to an SPU as needed. Choosing the Cell SPE to explore the use of garbage collection on a core with a local memory based memory hierarchy offered the advantage of using the open Linux-based Cell ecosystem [10].

Our contributions are as follows: 1) we identify the use of the local store and necessary synchronizations for offloading the mark-phase of the BDW collector to the coprocessor 2) we identify the necessary steps for managing coherency of reference liveness information between the host processor and the SPU, 3) we quantify the effects of using an MFC-based software cache to improve GC performance on the coprocessor giving a speedup of an average of 200% over accelerator baseline code without the software cache, 4) we demonstrate hybrid caching schemes adapted to the behavior of different data types and their reference behavior and demonstrate them to deliver a speedup of an average of 400% over the baseline, 5) we quantify the effects of previously known GC Prefetching strategies on the coprocessor that uses DMA for memory accesses 6) by extending the software cache, we implement Capitulative Loads using the MFC's DMA controller and quantify its performance impact on the coprocessor.

This paper is structured as follows: we describe workload characteristics of mark-and-sweep garbage collectors in section 2, and we describe the Cell SPE's local store and memory flow controller-based architecture in section 3. Section 4 gives an overview of the challenges involved in porting a garbage collector to a processor using a local memory as its primary operand storage, and we describe the use of software caches in section 5. We compare GC marking performance between a SPE and a PPE in section 6. We explore the use of prefetching techniques for garbage collectors executing in a local memory in section 7. We analyze coherence and data consistency issues in a local store-based co-processor in section 8. We discuss related work in section 9 and draw our conclusions in section 10.

2. Workload Characteristics

Garbage collection is in many aspects the antithesis of the application spaces for which local stores are developed and are intuitively useful. As such, this application is not the right application space to explore the performance of local-store based systems. Rather, porting such an application shows how well an application can run in an environment that makes no concessions to it, and analysis of porting challenges and solutions may allows us to better understand this architecture class.

Local stores are typically architected as data repositories for processor local data or as a staging ground for partitioned large dense data sets. Storing local data in a local store is attractive, because the local store guarantees access with guaranteed low latency, and without the overhead of coherence protocols when executed in a multiprocessor. For data-intensive applications, local stores offer an ideal repository for partitioned dense data sets, such as found in linear algebra computations used in many numerical applications. Eichenberger *et al.* [7] summarize the use of local stores with tiling and data set partitioning for these algorithms, and with a large computation to data transfer ratio. Similarly, Hwu *et al.* [15] describe the use of local stores for GPU computing.

Garbage collection represents the very opposite end of the application space, chasing pointers across a large memory space, with an infinitesimally small compute to data transfer ratio, and non-existing locality.

The Boehm-Demers-Weiser (BDW) mark-(lazy)sweep collector is popular due to its portability and language-independence. It epitomizes a class of collectors known as ambiguous roots collectors. Such collectors are able to forego precise information about roots and knowledge of the layout of objects, by assuming that any word-sized value is a potential heap reference. Any value that ambiguously appears to refer to the heap (while perhaps simply having a value that looks like a heap reference) is treated as a reference and the object to which it refers is considered to be live. The upshot of ambiguity is that ambiguously-referenced objects cannot move, since their ambiguous roots cannot be overwritten with the new address of the object; if the ambiguous value is not really a reference then it should not be modified. The BDW collector treats registers, static areas, and thread activation stacks ambiguously. If object layout information is available (from the application programmer or compiler), then the BDW collector can make use of it, but otherwise values contained in objects are also treated ambiguously.

The advantage of ambiguous roots collectors is their independence of the application programming language and compiler. The BDW collector supports garbage collection for applications coded in C and C++, which preclude accurate garbage collection because they are not type-safe. BDW is also often used with type-safe languages whose compilers do not provide the precise information necessary to support accurate GC. The minimal requirement is that source programs not hide references from GC, and that compilers not perform transformations that hide references from GC. Thus, BDW is used in more diverse settings than perhaps any other collector. As a result, the BDW collector has been heavily tuned, both for basic performance, and to minimize the negative impact of ambiguous roots [2].

The basic structure of mark-and-sweep garbage collection is depth-first search of all reachable pointers on the heap. For this purpose, an initial set of root pointers (from the application's register file, application stack, and known roots in the data segment) are used to find references into the application's heap. This is accomplished by initializing a mark stack with these known roots.

The mark phase removes heap addresses from the mark stack, and uses the reference in conjunction with information about the object pointed to by the discovered pointer to find any pointers stored in this object. The minimum amount of information necessary about an object is its starting address and length, which can be obtained from the memory allocator. For such an object, any properly aligned data words could be legal pointers.

Any newly discovered legal heap addresses found in this way are then pushed on the mark stack and the reachable objects are marked in a mark array. The algorithm iterates until the heap is empty.

The following code fragment describes the algorithm in more detail (assuming just a single type of records identified by the record length):

```
while (ptr = pop_mark_stack())

  length = alloc_size(ptr);

  for (i=0...length)
    if (legal_ptr(ptr[i]) && ! marked(ptr[i]))
      mark (ptr[i]);
      push_mark_stack(ptr[i]);
```

Once the algorithm has traversed all reachable heap objects, the mark bits represent a bitmap of all reachable objects. All unmarked objects can be de-allocated using a linear sweep over the heap. The sweep can be performed eagerly, or lazily at allocation request. Lazy sweep has preferable cache behavior because it avoids touching large amounts of cold data for the sole purpose of de-allocation [2].

Tracing GC schemes can be classified into stop-the-world and incremental. Stop-the-world GC suspends the mutator until a full pass of the GC is done, thus disallowing any change to the heap space during collection. Incremental GC allows interleaving mutator and GC, either in a sequential or parallel fashion. thus providing the benefit of concurrency, but at the cost of tracking liveness coherency between the mutator and the GC.

Our implementation is based on a stop-the-world approach where the mutator executes on the PPE, and when GC invokes the marking code, control is transferred to the mark code executing on the SPE. During this time, the mutator is suspended on the PPE and the PPE becomes available for other processes in the system and increase

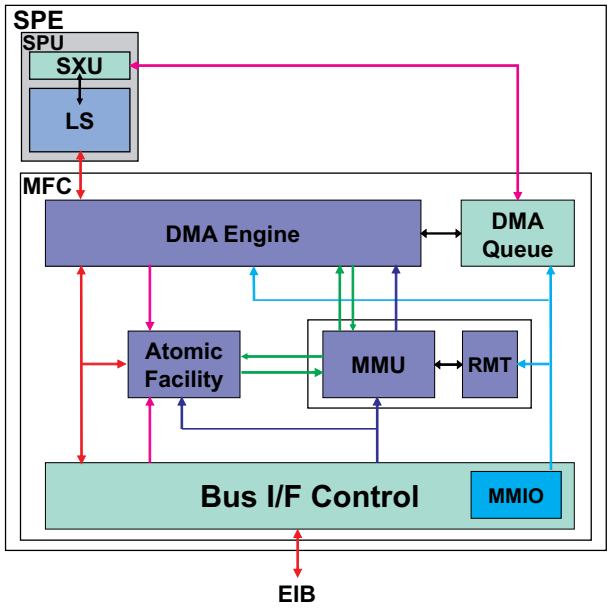

Figure 1. The Memory Flow Controller (MFC) implements the SPE's interface to the system memory. It consists of a DMA engine, memory management unit, and an atomic access facility.

overall system throughput. In this environment, the single-SPE GC can take advantage of all eight SPEs, when the host processor is running multi-programmed Java workloads and each program uses a dedicated single SPE for its garbage collection.

As a future addition, the same dirty-blocks tracking support used for maintaining mutator/GC coherency for incremental GC can be applied to SPE-based GC, and the PPE could continue to execute the mutator during the time freed up by SPE-based marking. In this scenario, because the mutator is rarely suspended to perform GC, concurrency already exists and no incrementality is needed in the GC.

3. Cell SPE Local Store Usage

In the Cell Broadband Engine, data is transferred to and from the local store using the synergistic memory flow controller. The memory flow controller (MFC) operates in parallel to the SPU execution unit of the SPE, offering independent computation and data transfer threads within each SPE thread context [9].

The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The SPU is a RISC-style processor with an instruction set and a microarchitecture designed for high-performance data streaming and data-intensive computation. The SPU includes a 256-Kbyte local-store memory to hold an SPU program's instructions and data. The SPU cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into local store or write computation results back to main memory. The SPU can continue program execution while the MFC independently performs these DMA transactions. No hardware data-load prediction structures exist for local store management, and each local store must be managed by software.

The MFC performs DMA operations to transfer data between local store and system memory. DMA operations specify system memory locations using fully compliant PowerPC effective addresses. DMA operations can transfer data between local store and any resources connected via the on-chip interconnect (main memory, another SPEs local store, or an I/O device).

In the PPE, effective addresses are used to specify memory addresses for load and store instructions of the Power Architecture ISA. On the SPE, these same effective addresses are used by the SPE to initiate the transfer of data between system memory and the local store by programming the MFC. The MFC translates the effective address, using segment tables and page tables, to an absolute address when initiating a DMA transfer between an SPE's local store and system memory.

In addition to providing efficient data sharing between PPE and SPE threads, the MFC also provides support for data protection and demand paging. Since each thread can reference memory only in its own process's memory space, memory address translation of DMA request addresses provides protection between multiple concurrent processes. In addition, indirection through the page translation hierarchy allows pages to be paged out. Like all exceptions generated within an SPE, page translation-related exceptions are forwarded to a PPE while the memory access is suspended. This allows the operating system executing on the PPE to page in data as necessary and restart the MFC data transfer when the data has been paged in.

MFC data transfers provide coherent data operations to ensure seamless data sharing between PPEs and SPEs. Thus, while performing a system memory to local store transfer, if the most recent data is contained in a PPE's cache hierarchy, the MFC data transfer will snoop the data from the cache. Likewise, during local store to system memory transfers, cache lines corresponding to the transferred data region are invalidated to ensure the next data access by the PPE will retrieve the correct data. Finally, the MFC's memory management unit maintains coherent TLBs with respect to the system-wide page tables [1].

While the MFC provides coherent transfers and memory mapping, a data transfer from system memory to local store creates a *data copy*. If synchronization between multiple data copies is required, this must be provided by an application-level mechanism.

MFC transfers between system memory and an SPE's local store can be initiated either by the local SPE using SPU channels commands, or by remote processor elements (either a PPE or an SPE) by programming the MFC via its memory mapped I/O interface. Using self-paced SPU accesses to transfer data is preferable to remote programming because transfers are easier to synchronize with processing from the SPE by querying the status channel, and because SPU channel commands offer better performance. In addition to the shorter latency involved in issuing a channel instruction from the SPU compared to a memory mapped I/O access to an uncached memory region, the DMA request queue accepting requests from the local SPU contains 16 entries compared to the eight entries available for buffering requests from remote nodes. Some features, such as the DMA list command, are only available from the local SPE via the channel interface.

Performing an actual transfer requires two communications events, first to indicate the SPE is ready to receive data because it has completed the previous work assignment, and as second synchronization to indicate the completion of a PPE-side transfer. From a programming point of view, SPE-initiated DMA requests are preferable because they reduce the need for double handshake communication, channel accesses to the MFC are cheaper, and because in parallel programs, they prevent the PPE from becoming a bottleneck.

This decision is reflected in the sizing of request queues, where each MFC has 16 transfer request entries in its queue reserved for the local SPE, and another 8 entries accessible by the PPE and remote SPEs via memory-mapped I/O registers.

The MFC also implements inbound and outbound mailbox queues. These mailbox queues are accessible with channel instructions from the local SPE, and using memory-mapped I/O registers from the PPE and remote SPEs.

Because each SPE comes with a limited local store size of 256KB for both instructions and data, we cautiously optimize our data structures with this limit in mind. In the final version, the use of the local store includes less than 20KB instruction image (including the

garbage collection code, management code for data caching and synchronization, and libraries), a 128KB software cache for caching heap and miscellaneous references, 40KB of header cache, 32KB of local mark stack, and a small activation record stack.

4. Porting GC to Processors with Local Store Hierarchies

In order to select a realistic optimized starting point which might serve as reference on a traditional processor, we chose the publicly available Boehm-Demers-Weiser garbage collector. We chose Java as the environment for our applications using the open-source GNU Compiler for Java (gcj).[1] To report garbage collection results, we use the SPECjvm98 benchmark suite as the primary drivers for the garbage collection. We use the jolden programs to show differences between different application types where appropriate.

The application executes on the Cell PPE. We retain applications unmodified from the original PPE executable to present a comparable memory map, as we concentrate on GC execution in this work. We rebuild the GCJ run-time environment to include calls to a thin communication layer with the SPE.

The communication layer contains GC-specific code for communication between the PPE and the SPE, whereby the SPE obtains work descriptors from the PPE. The communication layer also contains code to load the SPE with the GC functions, initiate execution on the SPE, and perform synchronization between PPE and SPE.

During development of the SPE garbage collector, we keep the PPE collector fully functional. After each SPE mark activity, we compare the mark bitmap and the statistics on visited nodes to ensure full compatibility and correct operation with PPE marking during porting.

In porting the BDW collector to the Cell SPE, we concentrate on the application heap traversal of the mark phase where the bulk of the execution time is spent. The sweep phase has linear behavior operating on dense mark bits, and the use of a local memory to store portions of an allocation's freelist and mark bits is straightforward. We retain the management of the global mark stack, including the discovery of initial root objects, the handling of mark stack overflow and other special events (like blacklist management) on the Cell PPE. These functions interact heavily with the Java managed runtime environment for the application, and are best executed on the core executing the application. Local store mark stack overflow is handled by copying the mark stack back to the global mark stack.

Synchronization between PPE and SPE occurs via the mailbox interface which allows efficient transfer between PPE and SPE of small data quantities. We pass a descriptor address, which is used by the SPE to copy in a descriptor and the mark stack. Because the SPE uses stalling mail-box accesses, this model provides implicit synchronization.

Three of the key data structures for the BDW mark-and-sweep algorithm are:

- Mark stack (MS) that contains system addresses to heap blocks that are to be scanned.
- Heap blocks (HBLK) that need to be scanned.
- Header blocks (HDR) that contain information about elements in a specific allocation bucket, and a mark bit map for the memory block with word granularity (1b per word).

Of these data structures, references to the HBLKs, i.e., the allocated heap blocks that are scanned for legal pointers are by far the most frequent references. Because these three structures are frequently accessed in the system memory, their accesses dominate the communication between SPE and PPE. Therefore these structures become primary targets for optimizing GC performance in the SPE.

[1] This choice limits the applications which can be executed in our environment. For example, the DaCapo benchmarks cannot be compiled with GCJ.

```
while (ptr = pop_mark_stack()) <- read MS

  length = alloc_size(ptr);

  for (i=0...length)
    p = ptr[i];                  <- read HBLK
    if (legal_ptr(p))
      get_hdr(p);                <- read HDR
      if (not_marked(p))         <- read HDR
        mark (p);                <- write HDR
        push_mark_stack(p);      <- write MS
```

Figure 2. Annotated usage of system memory data in mark-and-sweep garbage collection in the basic mark code of section 2.

Figure 2 shows when these structures are referenced in the BDW collector. The size of mark stack can grow as the marking proceeds. Because of the depth-first-traversal nature of BDW, the size of the mark stack is roughly characterized by the depth of the application data structure and therefore is reasonably small.

Heap blocks basically contain data and pointers in the system memory that is accessible to the application for its computations. In a traditional processor, BDW marking aggressively optimizes for locality of references. For example, BDW only traverses live references as opposed to the entire application heap and accesses heap blocks in chunks of 128B which corresponds to the typical size of an L2 cache block. Because of the pointer-chasing nature of accessing heap blocks, the access patterns exhibit poor locality that can hardly be captured by hardware caches or hardware stream prefetcher. Most GC prefetching research including Boehm Prefetching [2] and CHV Prefetching [4] targets optimizing accesses to heap blocks for improving GC performance.

The header block is referenced in the marking loop for reading and writing the mark bitmap of a heap reference. An index structure is used to locate HDR blocks for each heap address. To enable fast lookup of frequently used elements, the BDW comes with an optional lookup table. Each header contains a bitmap that represents liveness for a continuous 4KB region. To avoid repeating marking work when chasing pointer chains in the heap, the mark bit of a reference is first read to determine if the chain has previously been chased.

A naive implementation of the mark phase might replace every system memory reference to access these data structures with a DMA transfer to access system memory and transfer the referenced data to the local store before each access. We use the performance of this naive configuration as the baseline to determine performance improvement obtained with different local store caching strategies.

SPE-side data structure traversal is possible because the SPE DMA shares a common view of an application's virtual memory map, and any valid pointer reference on the PPE will be equally valid and refer to the same memory location on the SPE to initiate a data transfer from the associated address. A common system memory view in the SPE accelerators is a significant enabler of programming flexibility – in a system where data cannot be "pulled" by the accelerator, and input working sets need to be "pushed" by the host CPU, this sort of acceleration that involves traversal of data structures is not possible. A "push" data model also puts more load on the central processor, and threatens to cause a bottleneck based on Amdahl's observations of program parallelization.

While DMA requests for each system memory access can be used for isolated references without locality, references to data structures that exhibit locality must ultimately exploit the fast local store memory hierarchy level to cache data references and exploit this reference locality, or suffer unacceptable performance.

The SPE maintains a local mark stack which contains 4K entries, with 32KB being comparatively small relative to the SPU local store. This local mark stack captures all mark-stack push and pop

operations in SPE local store. Data transfers between the local mark stack and the global stack occur when the local mark stack is empty, or when it overflows, to receive additional global-mark stack entries, or to return a portion of the overflow.

To capture locality for other references, caching strategies must be employed by the SPU program. The simplest caching structure to be maintained is a memory region containing homogeneous data, as used by numeric applications to tile matrices in local store. Thus, during garbage collection, instead of individually loading each word with an MFC request when scanning a block of pointers `ptr[i]`, the entire block can be retrieved by a single DMA request. We sometimes refer to this approach as operand buffering (much like one or more memory operands are cached in microprocessor registers), or caching of memory chunks. These operand buffers are explicitly maintained and each buffer is a distinguished individual copy of a certain memory block. Operand buffers are good for discernable operands which are individually handled, fetched and maintained independently, e.g., in distinct buffers allocated for this purpose. Likewise, updates to such a structure can be gathered in local store and commit with a single copy back operation.

The correspondence of data in the operand buffer to actual system memory is usually implicit, i.e., there may not be any dedicated address information maintained for the operand buffer. (In numeric code, the address may be derivable from an array base address and a loop index, and in garbage collection, it may correspond to a pointer popped from the mark stack which may no longer be maintained in a register once scanning of a memory block commences.)

A second structure may cache objects which are of like type, e.g., by collecting records in a temporary store as they are used, and making them available for subsequent references. These objects maintain a home location in main memory, so this use of local store is distinct from allocating private objects in the local store. We refer to this type of cache as "object cache".

Access to object cache structures can occur by lookup with a system memory address (i.e., where the system memory home location serves as index), or by a content-based lookup (e.g., to find a data type layout descriptor by finding a record associated with a specific type). Our implementation uses an object cache for HDR blocks, and access it with a hashed system-memory effective address.

Linear scan of memory blocks for pointers can be made more efficient by requesting a memory range corresponding to an object to be scanned for valid heap pointers with a single DMA request into an operand buffer. The block is only maintained long enough to scan for pointers. Figure 3 and Figure 4 show the performance impact of using an operand buffer for memory blocks being scanned by the garbage collectors as it traverses the the heap on the execution time for the sum of all mark phases in the workload. Using operand buffering for heap blocks results in a mean speedup of 150% and 300% for Jolden and SPECjvm98, respectively.

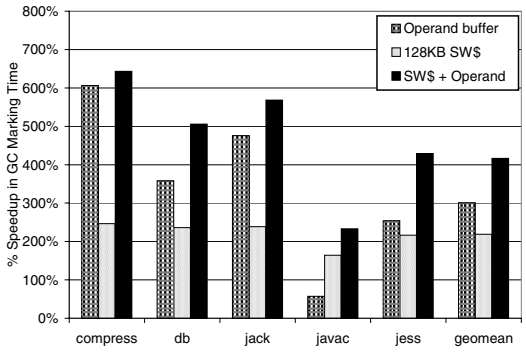

Figure 3. Garbage collection caching strategies for the SPECjvm98 benchmark. We compare the speedup obtained by operand buffering, a 128KB software cache, and combined software cache and operand buffering for large memory blocks over the initial baseline port.

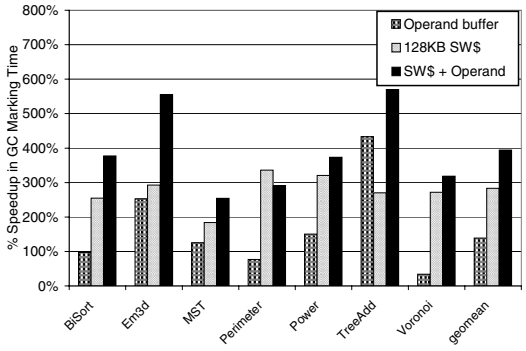

Figure 4. Speedup obtained with garbage collection caching strategy behavior for the Jolden benchmarks.

5. Software Caches

Operand buffers and object caches can handle contiguous data and homogeneous collections of important data structures, respectively, in a local store environment, but for many other references the locality may not be pronounced enough to support a scheme of storing a single data region for processing and then move to the next region.

To exploit locality in such circumstances, we use a "software cache". The software cache is a software abstraction to capture temporal and spatial locality in memory modeled on hardware caches, and a standard component of the Cell SDK distributed by IBM [7, 6]. Like hardware caches, there are a number of equivalence sets, indexed by a set of address bits, and blocks from the equivalence set being selected based on tag match checks.

Like hardware caches, equivalence sets are selected by a cache index formed from low order address bits. Using the 4-way SIMD capability, it is possible to efficiently implement a 4-way associative lookup. Software caches offer a catch-all fallback strategy for dealing with data sets – they offer retention between accesses based on address-based lookup, with data persisting in the local store beyond the use of region-copies used for dense references.

Software caches involve significant overheads in terms of instructions which must be *executed* by the SPU processor access latency to compute and compare tags, to determine a possible software cache miss, and to locate the data buffer serving as backing storage. Thus, for data with regular and predictable access behavior, software caches are a poor match. In porting applications, such data uses are best converted to employ operand buffers, or object caches. Using references to these objects which are known to reside in local store reduces the overhead of the software-based tag match. Software caches are most useful for large data sets with statistical locality, and where the benefits of a large cache can often outweigh the penalties of the long "hit latency" of a software cache.

Because software caches do not implement hardware coherence mechanisms we implement software application-level coherency support, which will be described in section 8.

Unlike hardware caches, software caches are not fixed in geometry, size, and replacement strategy at microprocessor design time. Instead, they can be adjusted to data sets and their behavior.

In porting our garbage collector, we have used the software cache provided with the IBM Cell SDK distributions.

Figure 3 quantifies the performance improvement in SPE mark time of a 128KB software cache with a 512B line size compared to the baseline design, and a pure operand buffering approach. While the application heap traversal only references each heap location only once (because marked blocks are not traversed again), the cache lines allow to exploit limited spatial locality with the prefetch effect of cache lines.

For large blocks, which may span one or multiple cache lines, no prefetch effect can be gained because they will not be co-located with other blocks, or not a sufficient number of blocks will share a line size for this effect to be effective. Thus, operand buffers are more beneficial for these accesses. To match and exploit the different behavior patterns, we design a hybrid caching strategy which partitions blocks into those using an operand buffer and those using the software cache. The "SW$ + Operand" of figure 3 this hybrid garbage collector using a software cache for small data references, and operand buffers for large heap blocks to be scanned. Using an operand buffer for large heap blocks to be scanned offers two benefits: (1) it reduces the hit latency by removing the access to the software cache tag store and the associated tag check code, and (2) it removes references with good, dense spatial locality, but little temporal locality, from the software cache to an optimized cache storing a region to be scanned.

Because each data reference's behavior is matched to an appropriate storage class in the local store, this hybrid implementation is the basis for the best performing SPE garbage collector. We use this hybrid configuration for all further experiments.

We refer to Figure 4 to demonstrate the dependence of the relative advantages of the different data caching schemes depending on the mix of allocated heap objects. Compared to the SPECjvm98 workloads, the jolden programs offers a higher fraction of small data blocks and show a significant advantage for the software cache over the operand buffer scheme. Like for the SPECjvm98 suite, the hybrid scheme performs best for the jolden benchmarks as well.

Figure 5 shows the impact of cache line size on software cache miss ratios during garbage collection for the SPECjvm98 benchmarks, using a hybrid caching scheme with operand buffer and software cache. As cache line size increases, the prefetching effect of cache lines reduces the miss ratio until cache lines of 512B. As the cache lines size is further increased, the small number of available lines at the constant cache size of 128KB leads to thrashing and a jump in cache miss ratios.

Figure 6 shows the impact of cache line size on performance of the software cache as speedup relative to the baseline. Because the software cache does not return the data to the application until the DMA concludes, the effect of transfer latency of larger blocks is more pronounced than in traditional caches (where critical word first deliver often allows applications to proceed while the trailing edge of a cache line is fetched, but subject to trailing edge bandwidth saturation). Thus, the miss ratio improvement of the 512B line over the 256B cache is compensated by the associated longer transfer latency, and resulting in a virtual performance tie.

6. Mark Performance Comparison between SPE and PPE

To understand the trade-off of offloading GC to the SPE, it is important to compare the GC performance of an SPE to that of a conventional microprocessor core. To that end, we perform quantitative comparison for GC performance of SPE to the PPE on the same chip. Because both SPE and PPE operate at the same 3.2 GHz frequency, the comparison highlights the architectural differences as well as the area efficiency of the two architectures.

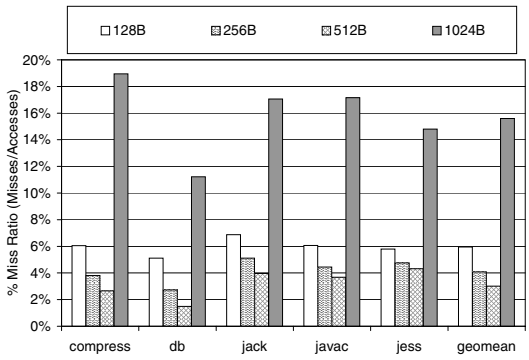

Figure 5. Miss Rates as a function of Line Size

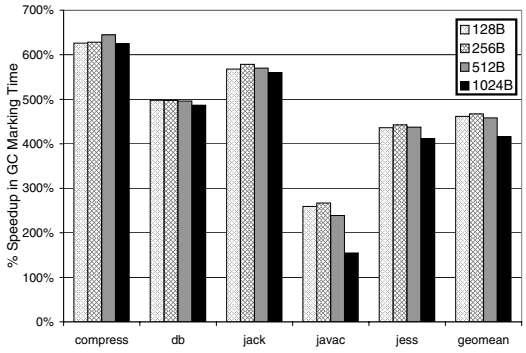

Figure 6. Software cache performance for different cache configurations (as speedup relative to the baseline).

Figure 7 shows GC's mark performance (a higher value represents better performance) of a PPE and an SPE using a 128KB software cache and an operand buffer, normalized to PPE's performance. We calculate SPE's mark performance using the formula 'PPE Mark Time / SPE Mark Time'. The SPE Mark Time includes the additional communication overhead of initiating and finalizing the offloading of GC from PPE to SPE including transferring of mark stack entries and writebacks of marks bits to the main memory also add to the SPE mark time. These results where obtained without SPE-specific code tuning, which can often deliver significant additional speedup. However, in this work our focus is on data management and optimization of data transfers, not SPE-specific code generation opportunities.

From Figure 7, we observe that SPE achieves reasonable GC mark performance compared to a PPE's performance for our workloads with a mean of 74.8% for the SPECjvm98 benchmarks and in Compress's case the SPE actually outperforms PPE mark performance by 4.7%. For Compress, achieving faster mark performance on SPE than PPE is possible because most of the accesses in Compress use the operand buffer which has shorter, fixed access latency than the average access latency of a PPE. For Db, Javac and Jess, SPE achieves 80.7%, 95.5% and 70.8% of PPE's mark performance, respectively. In Javac, SPE achieves 40.1% of PPE's mark performance. We attribute this lower performance to the larger live memory footprint and higher miss rate in the operand buffer.

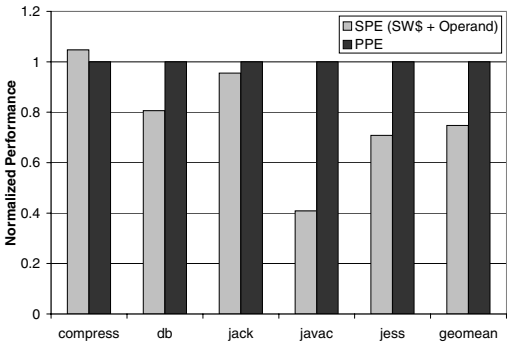

Figure 7. Mark phase performance comparison between PPE and SPE for the SPECjvm98 benchmark suite.

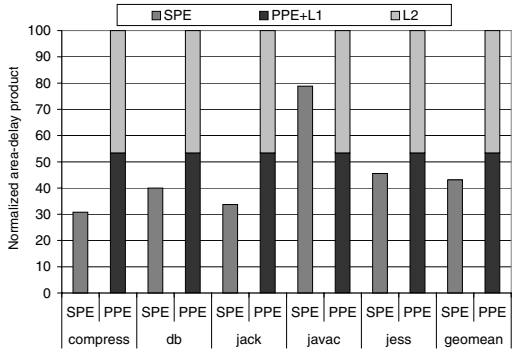

Figure 8. Area delay product comparison between PPE and SPE for the mark phase of the SPECjvm98 benchmark suite.

To understand better the performance difference between PPE and SPE, we examine their architectural differences in the following sections, as well as discuss some tradeoffs in the exploitation of explicitly managed memory hierarchies. The PPE uses split 32KB instruction and 32KB data level-1 caches while the SPE uses a local-store of 256KB for storing both instruction and data.

Because an SPE core is substantially smaller than a PPE core, it is also important to understand the mark efficiency by taking core area into consideration when comparing PPE and SPE. Core area is a proxy for silicon, power and cooling cost, as a larger logic area usually dissipates proportionally more power and heat. The area of an SPE core is 14.5 mm^2, whereas the area of a PPE core is 24mm^2. In addition, the Cell BE contains an L2 cache whose primary purpose is to service the PPE. When measuring GC mark performance / core area, SPE achieves 23% improvement in efficiency for SPECjvm98. If we consider the size of the L2 cache as well, the improvement in area efficiency is 132%. This comparison shows that although SPE was not designed for an application space such as garbage collection, we can achieve efficiency and performance that are comparable to a PPE through intelligent tuning of the application, thus greatly extending the possible use of Cell processors. By tuning code generation to take advantage of specific Cell SPE ISA features, the relative performance of SPE code can probably increased further beyond the SPE/PPE head to head comparison, but again, ILP/DLP optimizations are not the main focus of this work.

7. Application-Based Prefetching

Prefetching of data can be an effective strategy to handle large working sets with poor locality. Many advanced microprocessors include hardware prefetching engines.

Boehm [2] and Cher *et al.* [4] demonstrate the performance potential of application-directed prefetching for GC using prefetching instruction. Because of GC's non-strided, pointer-chasing nature, hardware-based stream prefetcher and out-of-order execution is ineffective for hiding GC memory latencies. In contrast, the garbage collector sends out prefetches for each request, and then buffers the requests to allow time for their prefetches to come back. While waiting for the prefetches, the garbage collector processes prior requests in the order the prefetches were sent out. Because CHV prefetcher exploits parallel branches in the data traversal, it does not traverse the mark stack in the same depth-first order as the original BDW.

In the interest of more aggressive application-directed long memory latency hiding techniques, Horowitz *et al.* [14] propose *informing memory instructions* in which the application can obtain the runtime hit/miss status of a cache access. Cher *et al.* [4] also show a special form of informing memory instructions (using simulators), known as *capitulative loads*, that becomes a demand-load on a cache hit and a prefetch on a cache miss, and returns the status to the application register. Simulated results from Cher *et al.* [4] show that capitulative load performs similarly to CHV prefetching for garbage collection in a traditional processor. CHV prefetching can be implemented as a special case of Capitulative Load where all dynamic instances of Capitulative loads always return cache miss; therefore, using capitulative load also lead to changes in access ordering.

Without the presence of hardware caches, traditional hardware memory-latency-hiding techniques such as out-of-order execution and prefetching are impossible. Hardware prefetching benefits from the presence of unarchitected and fast state storage. Prefetch instructions in this environment preload the target data into the cache to be quickly accessible in response to a memory access to the original load.

No binding of the data to a new address takes place in response to hardware prefetching. If the prefetch reference is illegal, no exception needs to be raised and the fetch can be suppressed. If the data is later accessed, and the reference is illegal, an exception will be raised at that point.

In comparison, prefetching data into the local store requires a copy operation which binds a local store address to the new data copy. Because the local store does not contain any validity indicator, the data must be assumed to be resident at the conclusion of the copy operation.[2]

Thus, prefetching data to the local store by explicit copy requires these copies to refer to correct addresses, and avoid speculation across writes that do not ensure single copy consistency.

One of the advantages of Cell is the ability to exploit application behavior to initiate prefetching of application data sets under application control by the DMA engine [20, 9]. Alas, the garbage collection with its irregular data patterns and unpredictable locality is the antithesis of this approach which is most efficient for regular dense data sets, e.g., with tiling of matrices.

However, we wanted to establish the potential of prefetching even in these challenging circumstances. In garbage collection, the prefetch of application heap blocks for use in future scan for point-

[2] Indicators of unsuccessful binding can be added to architected state, as used for speculative memory accesses in the IA64 architecture. Alas, this requires additional recovery code which is hard to test, and increases code size. If checking is implemented using explicit instructions, it also increases pathlength for the correct speculation case. If checking occurs implicitly, this may again introduce some of the critical timings that complicate traditional cache design.

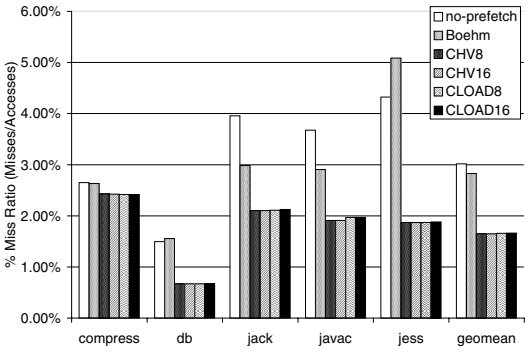

Figure 9. Miss Rates as a function of Prefetch strategy

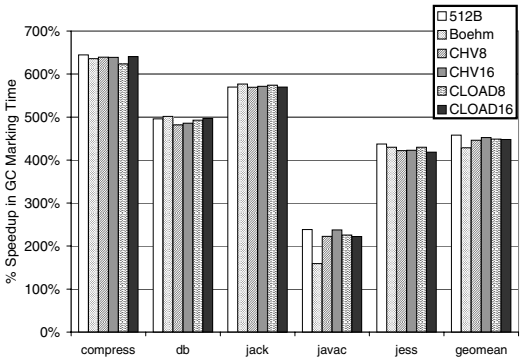

Figure 10. Prefetching Performance as speedup relative to baseline for no-prefetching (512B) and various prefetching strategies.

ers is the most promising approach. We can ensure the legality of prefetch by ensuring that addresses to be prefetched are within the heap bounds. In fact, the same condition is also the guarding condition for identifying a legal heap pointer to be traversed by the mark phase, and hence the prefetch aligns naturally with the garage collector application.

As mentioned before, because transfer time is a key concern for GC performance on SPE, we also experiment with three latency-hiding techniques for GC code running on conventional processors, namely Boehm Prefetching [2], CHV Prefetching [4] and Capitulative Loads [4]. All three schemes were shown to be effective for BDW GC on conventional processor but their effectiveness on processors with local stores have not been studied.

As mentioned before, the differences between prefetching on conventional versus on processor with local store are: 1) Addresses are binding, 2) Granularity of prefetching, 3) cost of misspeculation, and 4) cost of DMA.

We take these four differences into considerations when implementing prefetching for SPE. For example, each prefetcher checks if an address is within the legal heap boundaries before sending the corresponding prefetches. Similar to prefetching on hardware caches, the prefetcher sends out DMA requests for blocks that have the same alignment and cache line granularity as the software cache, but do not wait for its completion. When the actual demand arrives, the cache read code checks if the DMA has actually completed before consuming the corresponding data to avoid any data inconsistency.

Because Boehm Prefetching uses the mark stack for prefetching, it does not require a separate data structure for buffering prefetch addresses. For both CHV Prefetching and Capitulative Load, we use eight- and sixteen-entry buffers. Addresses are put into the tail of FIFO buffer when popped from the mark stack and consumed from the head of the buffer. As observed in [4], we observe reordering of marking accesses with both CHV Prefetching and Capitulative Load. For Capitulative Load, because the hit/miss status of the software cache are accessible by the GC code on SPE at run time, we experimented with reordering mark accesses according to hit/miss status in the software cache as suggested in [4].

Figure 9 shows the impact on miss ratio when GC is running on an SPE with various prefetching schemes on a 128KB software cache with a 512B line size. We fix the associativity at four-way across experiments in order to maintain a fixed, fast cache-lookup time for the software cache.

For the 128KB cache, Boehm Prefetching improves the miss ratio for Jack and Javac, but degrades the miss ratio for DB and Jess. Overall, the Boehm Prefetcher reduces Miss Rates by 0.2%. CHV Prefetching and Capitulative Load reduce the miss ratio more significantly for all benchmarks, overall reducing the miss ratio from 3.0% to 1.7%.

Figure 10 shows the impact on performance for these prefetching schemes. The graph is normalized against the base case that uses a 16B DMA transfer on each individual access. Surprisingly, the reduction in miss ratio does not seem to translate into performance gain. For the Boehm Prefetcher, because the miss ratios are not reduced as much, we conclude that the performance effect is minimal because the prefetches arrive early and are replaced in the software caches before they are used. For the CHV Prefetcher, the miss ratios are reduced but it does not improve performance. We conclude that the prefetches arrive so late that there is not enough independent work between a prefetch and its actual use to overlap for the miss latency. All schemes suffer minor performance degradation compared to not using any prefetcher because of pollution effects and instructions overhead for the prefetching.

8. Application-Level Coherence and Data Consistency Management

For uni-processors using the local store model exclusively, data consistency preservation is key when read/write data are referenced in different ways, e.g., using an object cache, and using explicit DMA to write, to avoid accessing stale data, or between heap references by the application and the collector.

For systems with multiple processors, e.g., a program employing a single PPE for general programs, and a single SPE, synchronization must be observed when handing off data between the processors.[3]

In our porting of the BDW garbage collector to the Cell BE, we have partitioned the application across a heterogeneous multiprocessor consisting of a PPE with a traditional cache-based memory hierarchy and an SPE with an explicitly managed local store. In partitioning the applications, we have retained the application program on the PPE to present a challenging application environment, and use the SPE for the bulk of the mark phase.

As a result of this partitioning in a multi-processor system with a local-store based accelerators, where application code and control code for the garbage collector operate on the PPE, synchronization of data accesses on PPE and SPE is necessary. In comparison, for applications running exclusively in a local store-based microprocessor, many of the synchronization operations discussed here would not be

[3] Fully parallelized programs across multiple local-store based processors require additional considerations which are beyond the scope of this work.

necessary because all memory accesses are performed by the same processor.

In the Cell Broadband Engine, all DMA transactions are coherent with respect to both PPE caches and TLBs and DMA operations and page translation hardware in the MFC participate in coherence traffic. However, data transferred into the local store are *copied*, and hence have the semantics of a copy. Consistent with copy semantics, when the original data source is modified, copies are not updated. When applications copy data, they must ensure that application code maintains copies consistent with respect to each other, and synchronize data when one copy is being updated.

For the BDW garbage collector, there are several types of data copied to the local store:

hdr blocks contain information about allocation buckets, and contain mark bits. Mark bits are modified by both the PPE-side and SPE-side mark algorithm. Synchronization must occur to prevent the use of stale HDR blocks. All modified HDR blocks must be flushed back to system memory after the end of the mark phase to be available during sweep.

bottom index is an index structure to locate the HDR blocks associated with each memory allocation area. The data is used read-only on the SPE to find HDR blocks. The structure is quasi-static structure and only changes when new allocation buckets are made available. Bottom index data copies only have to be changed at this time.

application's heap only changes during application execution. Data can be cached between multiple sub-invocations during the mark phase, this data is read-only for both SPE and PPE during garbage collection. Cached data copies must be invalidated when the application executes and can change contents.

To do an initial port of the garbage collection mark phase to the SPE, we used a simple consistency model – every time we handed off control from the SPE to the PPE, we invalidated all copies of data held in the SPE's local store.

This model guaranteed data consistency, because execution was not parallel between SPE and PPE, but sequential with SPE and PPE phases, and hence forcing data synchronization between SPE and PPE at transitions will cause all data references to be correct.

This implementation choice has a significant penalty, due to the large amount of data having to be repeatedly transferred. To address this, we added an application-managed data consistency scheme to reduce data copy synchronization cost. Alas, this scheme could not be a simple software implementation of hardware-based cache coherence which would require software messages to be generated on every PPE memory reference.

Instead, we turned to a scheme based on the data usage. Most hand-offs between SPE and PPE are due to the SPE having completed work on a portion of the mark stack, and the PPE assigning more work the SPE. Thus, in most instances of control hand-off between accelerator and core no data synchronization is necessary, except for the mark stack itself.

In our implementation with a PPE application incorporating lazy sweep, all HDR blocks must be flushed to their system home locations when the mark phase concludes for use by the lazy sweep. In implementations with an SPE-resident application, this synchronization is not necessary.

When the application changes its heap, cached heap objects may become stale. Because it is prohibitive to burden each heap reference by the application with a software coherence action, we invalidate the cached heap objects. As the heap objects are read only for the mark phase, this does not impose any additional traffic during invalidation, but will force an increased number of compulsory misses when a new mark phase commences.

On certain occasions the PPE performs PPE-local collection (such as when collecting objects with a private collector function identifying addresses referenced by an object), when the HDR objects must be synchronized. During these phases, the cached heap objects do not have to be invalidated because the application will not be active. Instead, only HDR objects need by synchronized.

Synchronization and coherence requests are handled via the Cell MFC mailbox function which allocates an incoming and outgoing mailbox queue to each SPE. To schedule a mark operation, the PPE sends the address of a work descriptor containing a portion of the local mark stack, as well as other parameters to the SPE. The mark stack and parameters are then transfered via DMA by the SPE, and a mark phase starts.

To support application-based coherence, the PPE can also send dedicated commands to write back or invalidate different caches, such as when the mark bits have been modified by the PPE-resident components of the collector, or when the heap has been modified by the application.

Finally, the SPE can return the contents of its mark stack when the SPE-local mark stack is too full.

9. Related Work

Boehm [3] was the first to apply prefetching techniques within GC, implemented within the Boehm-Demers-Weiser (BDW) collector. We begin with an overview of BDW before discussing how it incorporates prefetching.

The BDW collector supports a rudimentary form of prefetching during tracing. The basic approach is to prefetch the target of a reference at the point that the target object is discovered to be grey.

Having observed via profiling that a significant fraction of the time for this algorithm is spent retrieving the first pointer p from each grey object, Boehm [3] introduced a prefetch operation at the point where p is greyed and pushed on the mark stack. Boehm also permutes the code in the marker slightly, so that the last pointer contained in g to be pushed on the mark stack is prefetched first, so as to increase the distance between the prefetch and the load through that pointer. Thus, for every pointer p inside an object g, the prefetch operation on p is separated from any dereference of p by most of the pointer validity (necessary because of ambiguity) and mark bit checking code for the pointers contained in g. Boehm notes that this allows the prefetch to be helpful even for the first pointer followed, unlike the cases studied by Luk and Mowry using greedy prefetching on pointer-based data structures [17].

In addition, Boehm linearly prefetches a few cache lines ahead when scanning an object. These scan-prefetches help with large objects, as well as prefetching other relevant objects in the data structure being traced. Boehm reported prefetch speedups between 1% and 17% for a small set of C-based synthetic benchmarks and real applications.

Cher *et al.* [4] observe that because of the LIFO ordering in marking, PG sends out prefetch in different ordering than the respective use, thus causing prefetches to be either too early or too late. Cher *et al.* [4] improve upon PG by introducing Buffer Prefetching (BP). By using a software buffer to ensure both prefetch and use are in the same FIFO order, BP enables effective hiding of prefetching latencies with mark work and achieves better timeliness for prefetching. With a small four-entry buffer, BP shows 4% improvement in mark time on a real POWERPC 970.

Garner *et al.* [8] develop a methodology that performs thorough analysis on tracing using hardware performance counters on four different architectures. From the analysis, Garner *et al.* conclude that poor locality remains the principal bottleneck for tracing despite using prefetching techniques in [3, 4]. In addition, Garner *et al.* introduce Edge Order Traversal which improves 20-30% in garbage collection time for a large set of applications when combined with BP.

As described in section 2, GC represents the very opposite end of the application space for Cell's SPE processor. While our work attempts to optimize GC on existing Cell processors through software techniques, there is other work in the literature that proposes

GC-specific coprocessors to perform concurrent GC. [19] shows in simulations that threads that have very-low-cost communication (sometimes known as slices) can be used to offload reference counting using special hardware FIFOs between mutator and GC threads. Such hardware and fast communication do not exist in real processors today. Similar to our work, [13] proposes concurrent GC using special-purpose, smaller cores for GC. The main difference between our work and [13] is that their cores supports cache coherency and hardware support for profiling with the host processor, while our work is based on the existing Cell SPE that does not support cache coherency with the host processor. Therefore, we could not benefit from techniques described in [13]. The proposed hardware in [13] also does not exist today.

Gschwind [9] describes software pipelining of main memory requests with the DMA engine to exploit compute-transfer parallelism by overlapping computation and data transfer in an SPE thread. Williams *et al.* [20] describe the efficiency of explicitly managed memory hierarchies for high-performance computing workloads. Hwu *et al.* [15] explore explicitly managed memory hierarchies for GPUs.

10. Conclusions

We have shown the feasibility of mark-and-sweep garbage collection on processors with a local memory-based memory hierarchy. We have ported the BDW garbage collector to the Cell Broadband Engine with the mark phase executing on the Cell SPE. Executing service functions like garbage collection on a coprocessor allows better utilization of the main processor for executing application code.

We have explored caching of application data structures in local stores using garbage collection as a challenging application involving a variety of data access methods. We have shown that to optimize performance, different caching and buffering strategies should be employed to optimize overall performance.

Explicitly managed local stores give application programmers unprecedented flexibility and freedom to design caching structures for different data types in the application, to optimize each structure for the specific data reference behavior for a given data class. A single application can combine multiple of these structure-dependent buffering and caching schemes to exploit data-specific behavior and thereby maximize performance. We have demonstrated how to use a data reference-optimized strategy to achieve a performance gain of 400% up to 600% in managing data in a local store.

We have also analyzed local-stored based data prefetch and explored data prefetch in a local store environment with software-controlled cache for garbage collectors. We also experimented with an additional GC scheme which cannot be implemented on conventional processors but on processors with local store, namely software-controlled capitulative load. Finally, we have also explored application-based coherence management.

Based on our results, garbage collection with a local store hierarchy is a viable and competitive solution. Based on our results, we can exploit a lower complexity/power/area local-store based coprocessor to offload a critical runtime environment function and increase utilization of the main processor for higher value user application code. Based on our results, local stores give programmers unprecedented control over data placement in the memory hierarchy, and to optimize replacement and prefetching while ensuring efficient data migration with block-transfer-based DMA engines.

Acknowledgments

The authors would like to thank Mark Nutter, Kevin O'Brien, Sid Manning, Tong Chen, Kathryn O'Brien for their help and many useful discussions during the development of the Cell BDW garbage collector. The authors would like to thank Wen-mei Hwu, Daniel Prener, Steven Blackburn, Martin Hirzel, John-David Wellman, Dick Attanasio and Shane Ryoo for their many insightful suggestions in the preparation of this manuscript.

References

[1] Erik Altman, Peter Capek, Michael Gschwind, Peter Hofstee, James Kahle, Ravi Nair, Sumedh Sathaye, and John-David Wellman. Method and system for maintaining coherency in a multiprovessor system by broadcasting tlb invalidated entry instructions. U.S. Patent 6970982, November 2005.

[2] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Programming Language Design and Implementation (PLDI)*, June 1993.

[3] Hans-Juergen Boehm. Reducing garbage collector cache misses. In *ACM International Symposium on Memory Management*, 2000.

[4] Chen-Yong Cher, Antony Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In *Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[5] Scott Clark, Kent Haselhorst, Kerry Imming, John Irish, Dave Krolak, and Tolga Ozguner. Cell Broadband Engine interconnect and memory interface. In *Hot Chips 17*, Palo Alto, CA, August 2005.

[6] Alexandre Eichenberger et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine Architecture. *IBM System Journal*, 45(1), 2006.

[7] Alexandre Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter Oden, Daniel Prener, Janice Shepherd, Byoungro So, Zera Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the Cell processor. In *14th International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, September 2005.

[8] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In *International Symposium on Memory Management (ISMM)*, October 2007.

[9] Michael Gschwind. The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, June 2007.

[10] Michael Gschwind, David Erb, Sid Manning, and Mark Nutter. An open source environment for cell broadband engine system software. *IEEE Computer*, June 2007.

[11] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel SIMD architecture for the CELL heterogeneous chip-multiprocessor. In *Hot Chips 17*, Palo Alto, CA, August 2005.

[12] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, March 2006.

[13] Timothy H. Heil and James E. Smith. Concurrent garbage collection using hardware-assisted profiling. In *ACM International Symposium on Memory Management*, 2000.

[14] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: memory performance feedback mechanisms and their applications. *ACM Transactions on Computer Systems (TOCS)*, 16(2):170–205, May 1998.

[15] Wen-mei Hwu et al. Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX. In *Hot Chips 19*, Palo Alto, CA, 2007.

[16] James Kahle, Michael Day, Peter Hofstee, Charles Johns, Theodore Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, September 2005.

[17] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[18] Albert Noll, Andreas Gal, and Michael Franz. CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. Technical report, School of Information and Computer Science, University of California, Irvine, November 2006.

[19] Manoj Plakal and Charles N. Fischer. Concurrent garbage collection using program slices on multithreaded processors. In *ACM International Symposium on Memory Management*, 2001.

[20] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the Cell processor for scientific computing. In *ACM International Conference on Computing Frontiers*, May 2007.