Cache-Aware Many-Core Garbage Collection

Jin Zhou
University of California, Irvine
jzhou1@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

ABSTRACT

The wide-scale deployment of multi-core and many-core processors will necessitate fundamental changes to garbage collectors. Highly parallel garbage collection is critical to the performance of these systems — today's garbage collectors can quickly become the bottleneck for parallel programs. These processors will present additional new challenges — many contain non-uniform memory architectures in which some cores have faster access to certain regions of memory than other regions.

This paper presents a new cache-aware approach to garbage collection. Our collector balances the competing concerns of data locality and heap utilization to improve performance. We have implemented our garbage collector and present results on a 64-core TILEPro64 processor. Our cache-aware parallel collector speeds up garbage collection by up to 46.7×.

1. INTRODUCTION

With the wide-scale deployment of multi-core processors and the impending arrival of many-core processors, parallel garbage collection is becoming increasingly important to overall system performance. A simple application of Amdahl's law reveals that sequential collectors will become performance bottlenecks as the number of cores increases. Parallel garbage collectors have the potential to reduce garbage collection time, improve the scalability of applications, and improve overall performance.

Many parallel and concurrent garbage collection algorithms have been proposed [13, 10, 11, 8, 3, 18, 12]. This earlier work on parallel garbage collection was largely targeted towards symmetric multiprocessing systems in which all processors have uniform access to memory. Early many-core processors hint that the memory systems of future many-core processors will be significantly more complex than the SMP model and will require careful attention to memory management to maximize performance.

1.1 Microprocessor Trends

Researchers and microprocessor manufacturers have recently developed several many-core processors. Tilera ships the 64-core TILEPro64 microprocessor and recently announced a 100-core Tile-Gx processor [22]. Intel recently announced the experimental 48-core Single-chip Cloud Computer(SCC) processor [14] and has been developing the Larrabee processor [20]. Examining aspects of these early many-core processors provides useful insights into the designs of future mainstream many-core processors.

Processors will likely have memory systems that are increasingly non-uniform. As evidence, we consider an existing commercial many-core processor, the Tilera TilePro64 64-core processor. The TilePro64 has four memory controllers that are connected to the grid of cores through an on-chip network and a distributed caching implementation. Achieving optimal performance requires balancing memory accesses across the memory controllers and attention to caching strategies.

Mainstream processors have also already diverged from the uniform memory architectures provided by previous generations of processors. Both Intel Nehalem and AMD Opteron processors have integrated memory controllers — a consequence of this design is that a processor in a multiprocessor system has faster access to the memory banks connected to its local controller. The recent AMD Opteron Magny Cour processor has two memory controllers — cores have faster access to the memory banks connected to their local memory controller.

Moreover, as the number of cores increases we expect to see a transition away from today's broadcast-based cache coherence protocols towards distributed cache coherence protocols. Indeed, the TilePro64 processor includes a distributed cache coherence protocol. Each cache line in the TilePro64 processor has a home core which manages its coherence. An implication of this architecture is that writes to memory that is homed on the same core are less expensive than writes to remotely-homed memory locations.

It is likely that future processors will provide additional communication mechanisms beyond just cache-coherent shared memory. Both the TilePro64 and the SCC have a 2D-mesh network that connects the tiles. Both processors contain hardware support for low-latency message passing between cores.

1.2 Basic Approach

This paper presents a cache-aware, many-core parallel garbage collector. The collector is architected as a master/slave distributed system with a master core coordinating the actions of all other cores.

We have designed our collector to maximize memory locality. The underlying design principle is to have each core independently manage its own memory in the common case. Each core only garbage collects memory that is local to that core. On modern architectures this both eliminates extra

inter-cache traffic and can leverage improved performance that is available through the local memory controller on some processors.

Modern commercial garbage collectors combine many techniques to optimize performance including generational garbage collection, mark-sweep collection, and mark-compact collection. This paper focuses on mark-compact collection as this component of a modern collector proves challenging for both preserving locality and parallelization. The marking approach used in this paper can easily be extended to support parallel mark-sweep collection. Our cache-aware garbage collector can be used to optimize the collection of the old generation in generational collectors.

We expect that future many-core processors will provide low-latency communication mechanisms and our collector makes use of such mechanisms for coordinating the collection process and during the mark phase. However, the basic collector design does not rely upon these mechanisms and they can be replaced with queue structures for the mark phase and shared memory structures for coordination.

1.3 Contributions

This paper makes the following contributions:

- **Independent Collection:** It presents a garbage collector in which all cores independently mark and compact and only communicate when there is a reference from the memory collected by one core to the memory collected by another core.
- **Local Allocation:** When possible, our collector allocates memory that is local to the current core or its neighboring cores. This typically improves the performance of the application because the memory will be both locally cached and accessed through the nearest memory controller. Moreover, it minimizes inter-core communications during garbage collection.
- **Memory Organization:** It presents a new heap organization approach partitions the heap to support independent collection and manages heap fragmentation to support huge objects including objects that are larger than a heap partition.
- **Cache-Aware Garbage Collection:** It presents a garbage collector design that optimizes both memory usage and garbage collection for non-uniform memory architectures.
- **Network-Based Design:** It presents a garbage collector design that leverages the low-latency, on-chip networks of modern many-core processors for the mark phase of the garbage collector.
- **Evaluation:** It presents an evaluation of the garbage collector on a 64-core TilePro64 microprocessor for several benchmarks. The TilePro64 is a commercially available, many-core processor and likely representative of the many-core microprocessors that will become commonplace in the future.

The remainder of the paper is structured as follows. Section 2 presents a simple parallel garbage collection algorithm. Section 3 presents our cache-aware extension to the simple collector. Section 4 presents our extensions to support huge objects. Section 5 presents our evaluation of the approach on several benchmark applications. Section 6 discusses related work; we conclude in Section 7.

2. A SIMPLE PARALLEL COLLECTOR

We begin our presentation of our many-core garbage collector by describing a simple parallel garbage collector which is designed to scale well on many-core processors. We later extend this algorithm with a number of enhancements that make it practical for real applications.

The simple collector partitions the shared heap into disjoint partitions of equal size — there is one partition for each core. The core to which a shared heap partition is assigned is the *host core* for that shared heap partition. Each core only allocates memory from its own partition. When a partition runs out of free memory, garbage collection is triggered. The execution of all application threads (*mutators*) on all cores is halted.

Each core is also responsible for garbage collecting its own heap partition. It marks only objects in this partition and compacts the live objects within this partition — objects in the simple collector never leave the allocating core's heap partition.

Our collectors have a master/slave architecture. One core is statically designated as the master — it coordinates the phases of the collector and distributes large (megabytes) blocks of memory to the allocators on the individual cores. In our implementation, in addition to its coordination activities the master participates as any other core.

The collector is designed such that the coordination overhead is not significant. We have not observed any scaling issues in our experiments using the collector on a 64-core TilePro64 processor. We note that in these experiments, one core serves as both the master and a participant in the computation. If processors should ever become available with a sufficient number of cores to cause the master workload to become significant, dedicating a core to serve as the master is a simple way to further scale the implementation.

When a core does not have sufficient space in its local partition of the heap to process an allocation request, it sends a request to the master core to garbage collect the heap. The master core notifies all cores that garbage collection has been requested. The master core waits until all cores have acknowledged the receipt of the request and sends a message that they have reached a garbage collection safe point. The master then sends out a garbage collection start message to each slave core.

The algorithm follows the standard high-level phases for a mark-compact garbage collector:

1. **Mark Live Objects:** The collector marks the live objects.
2. **Compact Objects:** The collector next compacts the live objects into contiguous memory blocks.
3. **Update References:** The collector finally updates references to point to the new object locations.

Recall that our implementation uses low-latency messages on the on-chip network to coordinate the garbage collection process. Figure 1 presents a time line of all messages for reference. There is a line in the graph for the master core and two lines for slave cores. Arrows between the lines indicate messages. The symbol * indicates that the given message

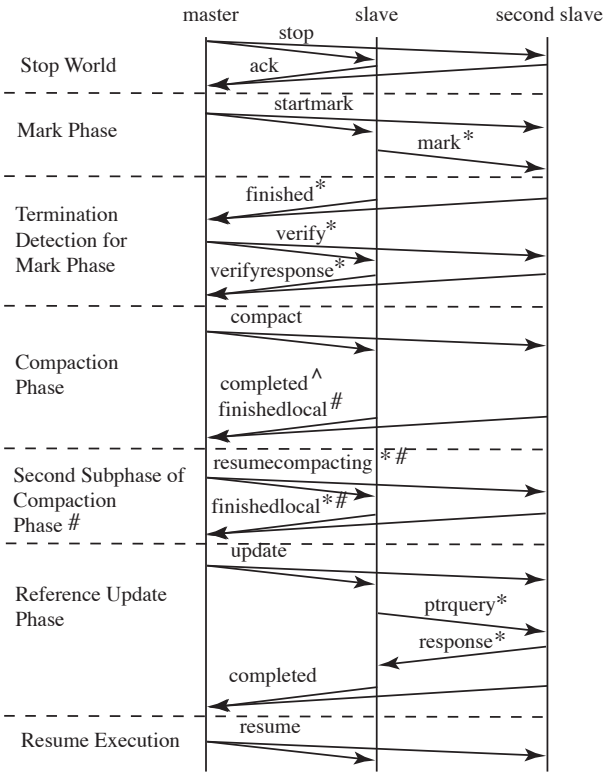

Figure 1: Messages Sent During Garbage Collection

can be sent repeatedly in the same garbage collection, the symbol ^ indicates that only the simple garbage collector uses the given message, and the symbol # indicates that only the cache-aware garbage collector uses the given message.

We next describe the garbage collection process in detail. When the master core receives notification that one core is out of memory, it sends a `stop` message to all cores. When it receives an `ack` message back from all cores indicating that all application threads have stopped, it sends a `startmark` message to all cores to begin the mark phase of garbage collection.

2.1 Mark Phase

Each core in the simple garbage collector marks the objects in its own heap partition. Each core runs its own collector, which maintains its own local mark queue of objects to mark. After a core receives the `startmark` message, the collector on the core begins the mark phase by scanning its local heap roots. When the core processes a reference, it checks whether the referenced object is located in the core's heap partition. If the referenced object resides in the local heap partition and it has not already been discovered, it is placed in the mark queue. References can of course span heap partitions; our collector uses the low latency on-chip network to communicate such references to the core that collects the partition containing the referenced object. If the object is located in another core's heap partition, the collector sends a `mark` message with the reference to the core that owns the partition that contains the object. When a core receives a `mark` message from another core, it checks whether the object has been marked. If not, it adds the object to the mark queue and records the mark request in a separate list. This list serves to track cross core references that must be updated. The list is only used to improve efficiency when updating references. We therefore fix the maximum size of the list; mark requests beyond this size are not recorded.

Each core executes a loop that dequeues a reference to an object from the mark queue, and then scans all of the references in that object. When the mark queue is empty, the collector on the core sends a `finished` message to the master. We next discuss the algorithm for detecting the completion of the mark phase.

2.1.1 Detecting Termination of Mark Phase

Determining that the mark phase has completed is nontrivial. The complication is that even after a core has completed scanning all live objects that it knows about, a `mark` message that contains the address of another live object in its partition can arrive. This problem is similar to the well-known problem of detecting termination of distributed algorithms [7]. We use the following algorithm to detect when the mark phase has terminated:

1. Each core sends a `finished` message to the master whenever it has completed scanning all known live objects. The message includes the number of `mark` messages it has sent and the number it has received. Note that a core can send multiple `finished` messages to the master.

2. When the master has completed scanning all known live objects in its partition, it checks a necessary condition for termination: that it has received `finished` messages from all slave cores and that the total number of `mark` messages sent matches the total number received. This condition alone is not sufficient to ensure termination, as the collected information does not necessarily represent a snapshot of the system's state. The master stores the send message and received message counts and then sends a `verify` message to all slaves.

3. When a slave core receives a `verify` message, it responds to the master with a `verifyresponse` message that includes (1) whether the core has completed scanning all known live objects and (2) the number of `mark` messages it has sent and number it has received.

4. Once the master has received all responses, it checks that (1) all cores report that they have completed scanning all objects and (2) that the total sent and received message counts match the previously stored totals. If both conditions are satisfied, the mark phase has terminated. Otherwise, the master restarts the algorithm whenever it receives a new `finished` message.

The correctness of this algorithm is straightforward. We note that a core that has finished scanning all of its known live objects can only become active upon receipt of a `mark` message. Verifying that the total number of received `mark` messages is the same ensures that each core remained inactive during the time period from when it sent its final `finished` message until it sent its `verifyresponse` message. Because the master receives all `finished` messages before sending the first `verify` message, the collected information represents a valid snapshot of the system during the instant between receiving the last `finished` message and before sending the first `verify` message.

2.1.2 Reference Caching

If an object is referenced by many other objects, the core hosting the partition that contains the object can potentially receive a large number of `mark` messages for the object. Each core therefore implements a fixed-size hashtable that caches recently sent object references. If a reference to a remote object hits in this cache, a `mark` message has already been sent for this object and the core elides sending a `mark` message for the reference.

2.2 Compaction Phase

When the master collector detects that the mark phase has completed, it sends all slave cores a `compact` message to inform them to start compacting the heap.

When a core receives the `compact` message, it linearly traverses its local heap partition and slides objects forward to compact the heap. When it moves an object, it records the object's new location in a table.

After a core completes compacting its heap, it builds a hash table from the old locations to the new locations for the objects for which it received `mark` messages using the stored list. This table is stored in shared memory and made available to other cores. Finally, the core sends a `completed` message to the master core to indicate that it has completed the compaction phase.

2.3 Reference Update Phase

When the master has received `completed` messages from all cores and has finished compacting its local heap, it sends an `update` message to all cores. When a core receives this message, it begins to update the object references in its heap partition.

Each core begins by updating the object references in its runtime data structures and then updates the object references in its partition of the heap. When it finishes the update phase, it sends a `completed` message to the master core.

Each core checks if a reference is to its local heap partition, and if so computes the updated reference using its local table. If a reference is to a remote heap partition, the core looks up the reference in the remote cores shared table. If the reference is not in that table, the core sends a `ptrquery` message along with the old location to the remote core which responds with a `response` message and the new location.

An alternative implementation is to store all tables in shared memory. We used local memory for these tables, because we want to make nearly all of the 32-bit address space available for use for the shared heap. An alternative approach for 64-bit processors is to simply use a 64-bit address space.

When the master has received `completed` messages from all cores and has finished updating its references, it sends a `resume` message to all cores. When a core receives this message, it resumes execution of the application thread.

3. CACHE-AWARE COLLECTOR

The simple parallel collector is designed to scale well on many-core processors. However, the heap partitioning scheme poses problems for many applications. The partitioning scheme fragments the heap into many small pieces, and therefore may make it impossible to allocate huge objects even after garbage collection. Moreover, the scheme has a problem for applications with imbalanced allocation rates across cores. If one core allocates significantly more memory than the other cores, it can run out of memory even when plenty of memory exists in the heap. This can cause excessive garbage collection or even cause the program to effectively run out of memory. In this section, we describe how we extend the simple collector to support a wide range of workloads while preserving parallelism.

The presentation of the cache-aware collector is organized as follows. Section 3.1 describes how the collector partitions the heap and maps partitions to cores. Section 3.2 presents the basic collection algorithm. Section 3.3 describes the how the collector manages fragmentation across cores. Section 3.4 presents the allocation algorithm.

3.1 Heap Mapping

We partition the heap into N partitions, where N is significantly larger than the number of cores C. The reason for creating many more partitions of the heap than the number of cores is to provide the allocator with the flexibility to allocate memory where it is needed while still maintaining locality. The cache-aware collector allows heap partitions to vary in size with the constraint that they must be an integer number of pages.

Each partition has a host core. In general the partition is mapped to a host core according to memory locality concerns for the given processor. In our implementation, memory partitions are mapped such that the cache lines for the memory partition are homed on the partition's host core and that the physical addresses correspond to the closest memory controller. The host core is responsible for marking and compacting the partition. While our collector preferentially allocates objects into the local partition, it will allow cores to allocate objects into other partitions when necessary to balance heap usage or to support huge objects.

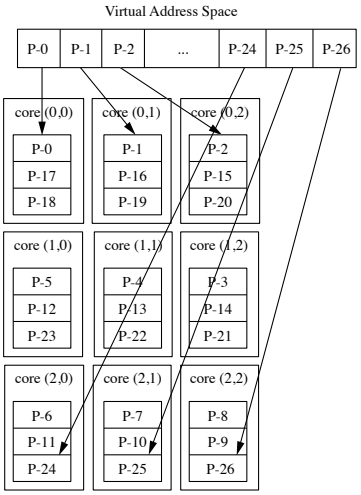

Figure 2: Heap Mapping

We next discuss how the heap's partitions are mapped onto cores. Figure 2 presents a mapping of the shared heap partitions to the cores of a 3×3 2-D mesh multi-core processor with $N = 27$ partitions. The mapping stripes the address space across the cores. The mapping ensures that partitions that are adjacent in virtual address space are hosted

on nearby cores.

The ubiquitous use of virtual memory systems in modern processors means that the memory controller associated with a heap partition is orthogonal to the heap partition's virtual address. Our garbage collector ensures that each heap partition has physical memory addresses that correspond to the nearest memory controller. This serves to both reduce traffic on the on-chip network and to balance load across all memory controllers.

3.2 Garbage Collection

We next discuss the cache-aware garbage collection algorithm. Traditional mark-compact collectors globally compact objects towards the bottom of the heap. Therefore, the compaction phase in traditional mark-compact collectors is inherently sequential and must be modified for parallelization. One challenge is that the compaction phase reads from locations that it later overwrites; Li et al present a parallel compacting algorithm that manages these dependences to safely compact the heap [15].

A more fundamental problem is that the standard approach to heap compaction is poorly suited for the memory systems of many-core processors. Traditional compaction moves objects to new memory locations that are potentially located on different memory controllers and whose cache lines are homed on other cores. This means that thread-local data is likely to be migrated to memory that is not on the closest memory controller nor whose caches lines are homed on the local core. This will likely both increase the garbage collection time and slow down the execution of the actual program code. An additional downside is that this migration makes the copy operations more expensive during garbage collection.

Our garbage collector takes a fundamentally different approach. The basic idea is that each core compacts the heap within the heap partitions that are hosted locally on core when possible. The arrows in Figure 3 shows the strategy for the example heap mapping. The strategy preserves locality while still generating large contiguous free blocks of memory that are suitable for huge objects at the top of the heap.

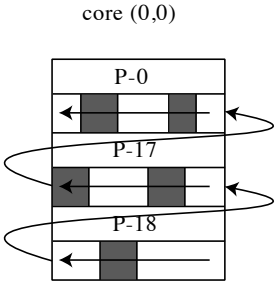

Figure 3: Garbage Collection Strategy

3.3 Managing Fragmentation Across Cores

Our locality-preserving garbage strategy can lead to fragmentation at the heap partition granularity in the shared heap. Figure 4 presents an example fragmented heap. The problem is that some cores may have more live objects and as a result fill their highest partitions while other cores have space left in their lower partitions. The resulting heap then does not have large contiguous free blocks of memory available for huge objects. Additionally, some cores are left with no free space in their local heap partitions to allocate new objects.

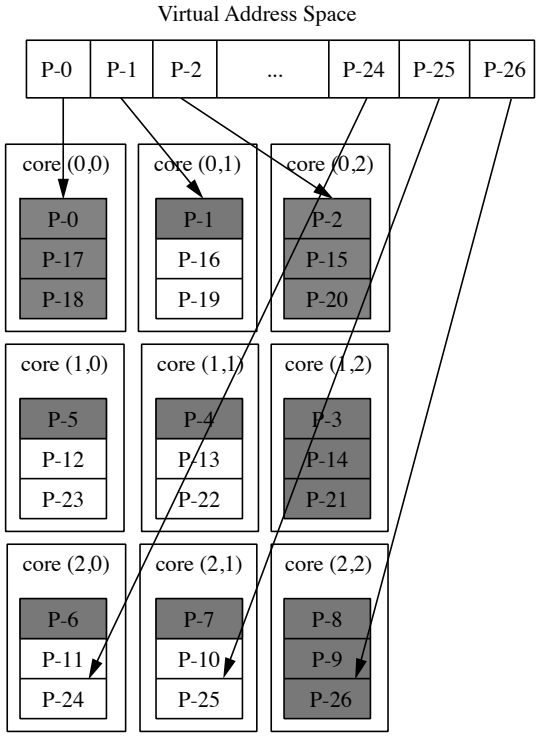

Figure 4: Fragmentation Problem

Our collector uses a partition balancing algorithm in the compaction phase to minimize cross-core fragmentation and to ensure that all cores have some free memory in their local heap partitions. During the mark phase, each core computes the total size of the live objects in its heap partitions and sends this information to the master collector. When the mark phase completes, the master collector uses these sizes to estimate the average number of heap partitions each core will fill during the compaction phase.

A core can locally compact its heap partitions up to this average number of heap partition limit without introducing fragmentation. We therefore use this average as an initial upper bound on each core's local compacting. We conceptually split the compaction phase into two subphases: the local compaction subphase and the balancing subphase. We note that this split is not global — while an individual core must be in one of the two subphases, different cores can simultaneously be in different subphases.

We first describe the local compaction subphase. After the mark phase has completed, the master collector in the cache-aware garbage collector sends each core a `compact` message that contains the initial upper bound on the compaction phase. Upon receiving this message, each core compacts its local heap until it either finishes or the newly compacted portion hits the initial upper bound. At this point each core responds to the master with a `finishedlocal` message that includes either (1) the total size of the objects remaining to be compacted if it did not finish compacting or (2) the top

of its locally compacted heap if it finished compacting.

When the master core receives a `finishedlocal` message from a core that finished compacting, it records the top of the heap in the table. When it receives a `finishedlocal` message from a core that ran out space, it searches this table to find space. It begins by first searching the table entries for neighboring cores for space. If space is not available from the neighboring core's heap partitions, it performs a global search in the table. When it locates space it sends a `resumecompacting` message along with the address of the destination partition and the number of free bytes remaining in the partition. The core then resumes compacting. When it finishes compacting, it sends another `finishedlocal` message as in the first subphase.

We note that in general, cores that respond in the first subphase with a `finishedlocal` message that they have extra space will typically complete compacting first as they have less work to do. Therefore, we expect that `finishedlocal` messages that request more space will typically arrive later, when space is already available. If a `finishedlocal` message requesting more space arrives before space is available and some cores have not responded to the first subphase, the master simply waits for other cores to respond to provide extra space. If space is still unavailable even after the other cores respond, the master will send a `resumecompacting` message along with a pointer to the first free block of memory in the heap.

3.4 Two-Level Memory Allocator

The challenge for the allocator is to manage both data locality and heap fragmentation. While the allocator for the simple parallel collector manages locality well, it fragments the heap preventing the allocation of huge objects and it poorly utilizes the heap potentially causing programs to unnecessarily run out of memory. While a traditional mark-compact allocator avoids fragmentation and heap utilization problems, it ignores memory locality concerns that are critical for performance on many-core processors.

Our allocator uses a two-level design: the top-level allocator manages the competing concerns of data locality and heap utilization when allocating large blocks of memory while the second-level allocator efficiently allocates small blocks of memory at minimal overhead.

Each core has its own second-level allocator. The second-level allocator uses a modified version of the standard pointer increment allocator. One issue that becomes important with a large number of cores is clearing memory — it is better to clear memory when it is used and thereby spread the required data transfer over a longer time period. Moreover, many modern processors contain instructions that clear an entire cache line without reading from main memory. The second-level allocator uses these instructions to clear memory on demand in tunable size blocks — we found that blocks of 4,096 bytes yielded optimal performance. This both spreads memory clearing over a longer time period thereby lowering our use of off-chip memory bandwidth and avoids cache misses on newly allocated objects. The second-level allocator requests memory in large blocks from the top-level allocator by sending messages. The second-level allocator maintains up to two large blocks of memory that it allocates into to ensure that a huge object allocation does not cause the allocator to leave a large part of the previous block of memory unused.

There is a single top-level allocator that executes on the master core. The master core maintains a table that tracks all heap partitions in the system. It uses this table to allocate space to the second-level allocators. This top-level allocator uses the following allocation strategy to manage both locality and heap utilization concerns:

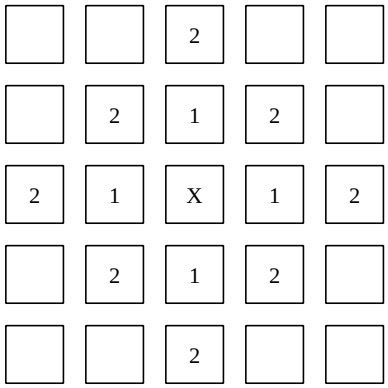

Figure 5: Neighbors

1. **Local Search:** The top-level allocator first attempts to give the second-level collector a block of memory from the local core's heap partitions. If the local core runs out of heap space, the collector falls back to the neighboring core search.

2. **Neighboring Core Search:** The top-level allocator next attempts to give the second-level collector a block of memory from one of the neighboring core's partitions. Figure 5 illustrates this strategy. The `X` marks the core that needs memory. The top-level allocator first searches for a free memory partition on one of the four nearest neighbors marked `1`. Within this group of partitions, it chooses the partition that is lowest in the heap. If no heap partition is found, it expands it search to include the next eight closest neighbors marked `2`. Within this group of partitions, it chooses the partition that is lowest in the heap. If the neighboring core search fails it falls back to the global search.

3. **Global Search:** The global search is designed for applications in which a handful of cores allocate nearly all of the memory in the program. The global search uses a heuristic to detect if the allocation rate is evenly distributed across all cores, and in which case it simply triggers a garbage collection. The heuristic checks if less than a tunable threshold of the heap has been allocated, and if so sets a flag to denote that global search should be used. If the flag is set, the global search returns the first free block of memory. If the flag is not set, the top-level allocator triggers a garbage collection event. Once set, the flag remains set until the next garbage collection event.

4. SUPPORTING HUGE OBJECTS

Our presentation of the cache-aware collector thus far has assumed that programs only allocate objects that are smaller than a heap partition. In this section, we discuss how our

collector supports objects that are larger than a heap partition. We call such objects huge objects.

4.1 Data Structures for Huge Objects

Each core collector maintains a local huge object list to track its local huge objects and the master collector maintains a global list to track all huge objects in the system. This list is always relatively short as the total number of huge objects must be fewer than the number of heap partitions in the system by definition.

During the compaction phase, a core collector scans its local heap partitions to compact all its live objects to the bottom of its heap partitions. The possibility arises that the beginning of a heap partition may contain the end or middle of a huge object that began in another heap partition. The compactor must recognize such heap partitions to skip over the space taken by huge objects.

Therefore, the cache-aware garbage collector maintains a *partition-head table*. This partition-head table tracks which blocks huge objects use. Each entry in the table specifies the start of the first normal object in the heap partition. If a heap partition is completely used by a huge object, the corresponding table entry is set to -1. Both the top-level allocator and the garbage collector update the partition-head table.

4.2 Additional Processes for Huge Objects

We next discuss how our collector handles huge objects during the compaction phase. One option is to simply leave the huge objects in place and compact the heap around them. However, our collector also supports compacting huge objects when necessary to eliminate heap fragmentation. The compacting algorithm performs the following steps:

1. Huge objects are marked as normal during the mark phase.
2. Huge objects are first cached at the top of the heap so that the space used by them can be used for compacting other live objects.
3. After all normal objects have been compacted, the huge objects are compacted from the top of the heap at the end of the compaction phase.

Compacting huge objects requires extra space at the top of the heap. The top-level allocator keeps track of the total size of all huge objects and triggers garbage collection if an allocation request would cause the free space in the heap to drop below the size of the existing huge objects.

Alternatively, huge objects could be left in place during the normal compaction phase and a final sequential compaction phase for huge objects could be used to eliminate the need for the extra free space. For some heap configurations, it is possible to parallelize this serial compaction phase.

4.3 Allocator Modifications

We make minor modifications to the previous allocation strategy. Huge objects require at least two contiguous heap partitions. The master core uses the same basic search strategy, but for each heap partition it examines it must also check whether enough contiguous heap blocks are free. If the allocator fails to find a large enough contiguous heap block, it forces a garbage collection event in which huge objects are compacted.

5. EVALUATION

We have implemented our garbage collector in our Java compiler and runtime system, which contains approximately 130,000 lines of Java and C code. The compiler generates C code that runs on the TILEPro64 processor. The TILEPro64 processor contains 64 cores interconnected through an on-chip network. The source code for our benchmarks and compiler is available on the web. We executed our benchmarks on a 700MHz TILEPro64 processor. We only used 62 of the 64 cores as 2 cores are dedicated to the PCI bus.

5.1 Benchmarks

Most of the benchmarks traditionally used to evaluate garbage collectors are sequential and generate relatively low allocation rates for a parallel collector. We therefore selected several sequential garbage collection benchmarks that were relatively straightforward to modify to present a highly-parallel, allocation-intensive workload. We parallelized the computation of the MonteCarlo benchmark; we modified the other benchmarks to simply execute multiple copies of the same computation. We have evaluated our garbage collector on the following eight benchmarks:

- **GCBench:** GCBench builds an initial set of long-lived data structures including a tree and an array and then builds several short-lived trees of various sizes. GCBench was originally written by John Ellis and Pete Kovac and then later modified by Hans Boehm [5]. GCBench is single-threaded, we modified GCBench to execute 62 threads each of which execute the original benchmark. We set the heap size for GCBench at 992 MB and modified the benchmark to maintain approximately the same heap utilization as the original sequential version.
- **Tree:** Tree continuously adds and removes nodes to and from a persistent binary search tree. We set the heap size for Tree at 310 MB.
- **FibHeaps:** FibHeaps performs insertions and removals on a Fibonacci heap. FibHeaps was ported from the Haskell version in nobench [1]. We set the heap size for FibHeaps at 310 MB.
- **LCSS:** LCSS implements Hirschberg's algorithm for finding the Longest Common SubSequence. LCSS was ported from the Haskell version in the nofib benchmark suite [19]. We set the heap size for LCSS at 310 MB.
- **MonteCarlo:** MonteCarlo performs a MonteCarlo simulation. MonteCarlo was taken from the Java Grande benchmark suite [21]. We set the heap size for MonteCarlo at 310 MB.
- **Voronoi:** Voronoi computes the Voronoi diagram of the two subtrees and merges them. Voronoi was taken from the JOlden benchmark suite [6]. We set the heap size for Voronoi at 1240MB.

- **BarnesHut:** BarnesHut performs a n-body simulation. BarnesHut was taken from the JOlden benchmark suite [6]. We set the heap size for BarnesHut at 310MB.
- **TSP:** TSP solves the traveling salesman problem. TSP was taken from the JOlden benchmark suite [6]. We set the heap size for Voronoi at 62MB.

We report results for the following garbage collection implementations:

- **base:** The base collector is a single-threaded mark-compact collector. Cache lines are homed on cores based on lower-order bits to evenly distributed memory requests.
- **dispar:** The dispar collector is a parallelized mark-compact collector. Cache lines are homed on cores based on lower-order bits to evenly distributed memory requests.
- **cacheaware:** The cacheaware collector is a locality-aware, parallelized mark-compact collector. Cache lines for each heap partition are homed on the core that hosts the heap partition.

5.2 Performance

Figure 6 presents speedups for our benchmarks. The speedups are given as the number of times faster than a single-core, sequential version of the benchmark. Figure 7 presents the percentage of execution time each benchmark spends collecting garbage. Our benchmarks were selected as they are very allocation intensive; therefore a large percentage of the work in our benchmarks is garbage collection. We expect that more typical applications will present a small garbage collection workload.

The base versions of the Tree, GCBench, FibHeaps, LCSS, Voronoi, and BarnesHut benchmarks do not scale. Closer examination reveals the reason is that the base versions of these benchmarks spend the vast majority of their execution time in the single-threaded garbage collector.

The dispar versions achieved a 5.2× speedup for Tree, a 15.0× speedup for GCBench, a 15.7× speedup for FibHeaps, a 27.6× speedup for LCSS, a 33.4× speedup for MonteCarlo, a 19.2× speedup for Voronoi, a 46.3× speedup for BarnesHut, and a 36.5× speedup for TSP. Figure 7 reveals the reason for the speedups — the dispar versions of our benchmarks spend a significantly smaller percentage of their execution time collecting garbage.

The cacheaware versions achieved a 46.0× speedup for Tree, a 27.9× speedup for GCBench, a 27.4× speedup for FibHeaps, a 39.1× speedup for LCSS, a 35.2× speedup for MonteCarlo, a 29.8× speedup for Voronoi, a 60.2× speedup for BarnesHut, and a 56.5× speedup for TSP. Figure 7 reveals that the cacheaware versions typically spend a smaller percentage of their execution collecting garbage with the exception of GCBench and Voronoi. The cacheaware version of GCBench and Voronoi spends a larger percentage in the garbage collector because the cacheaware version collector generates locality improvements that speed up the application threads by even more than the collector.

Figure 8 presents the speedup of the garbage collector. These numbers are presented as speedups relative to the base version's garbage collector. We omit comparison here to the sequential version of the benchmark because the garbage collection workload for the parallel and sequential bench-

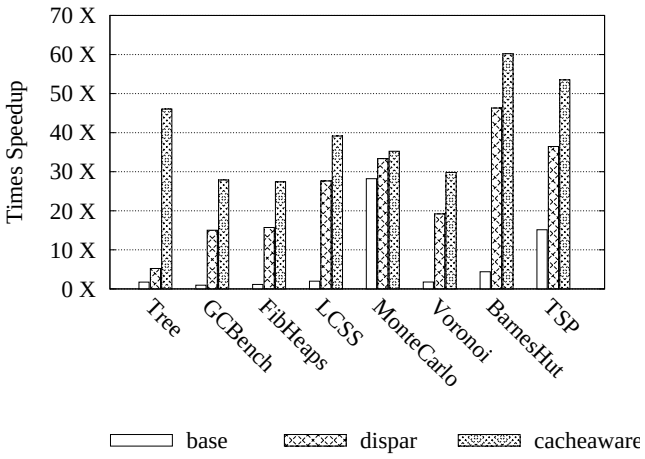

Figure 6: Overall Speedup

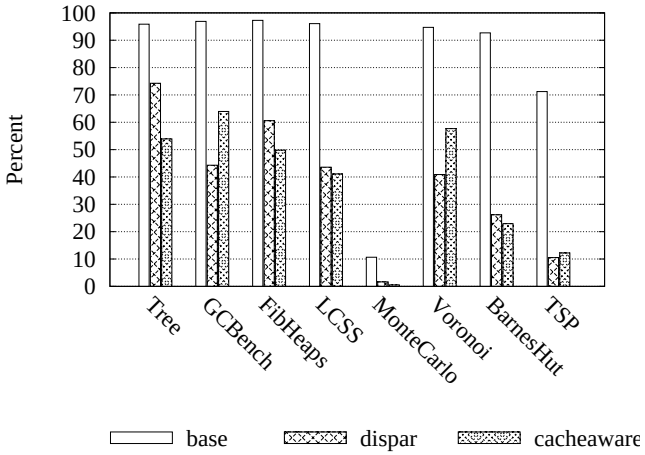

Figure 7: Percentage of Time Spent in GC

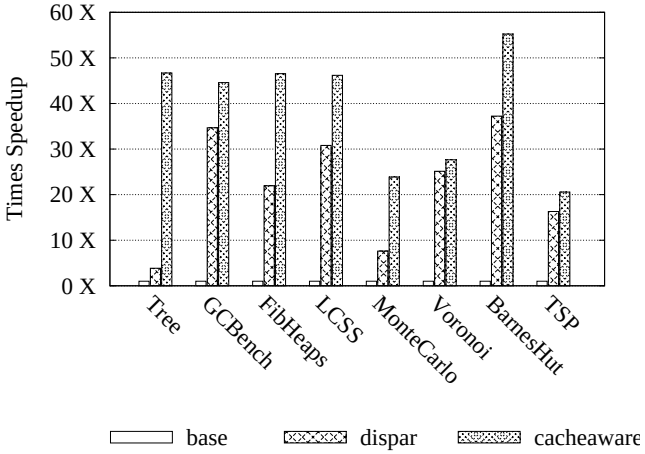

Figure 8: GC Speedup

marks are not comparable — the set of live objects is much (62×) larger for the parallel versions of most benchmarks. This means that the garbage collector must do more work to free less space. The garbage collection times reveal that the cacheaware garbage collector achieved significant speedups relative to the base collector for all benchmarks. Closer examination of the results for the dispar collector reveals that parallelism is responsible for significant fraction of the speedups. Comparing the results for the dispar collector with the results for the cacheaware collector reveals that homing cache lines on the core collecting the garbage significantly further improves performance.

Figure 9 presents the speedup of the mutator (the fraction of time spent in application code) relative to the sequential version. We note that cache effects can result in superlinear speedups; shared data can be cached by other cores in the 62-core version effectively increasing the size of the cache. The base versions and dispar versions of the benchmarks show similar speedups. The cacheaware versions show relatively larger speedups — the cache-aware collector allocates memory to ensure that most memory accesses are to memory whose cache lines are homed on the local core. This reduces cache coherence traffic and reduces the number of times cache lines must be copied from a remote core's cache. It can also effectively increase the size of the L2 cache as it reduces the number of cache lines that are only present to home data accessed by a remote core.

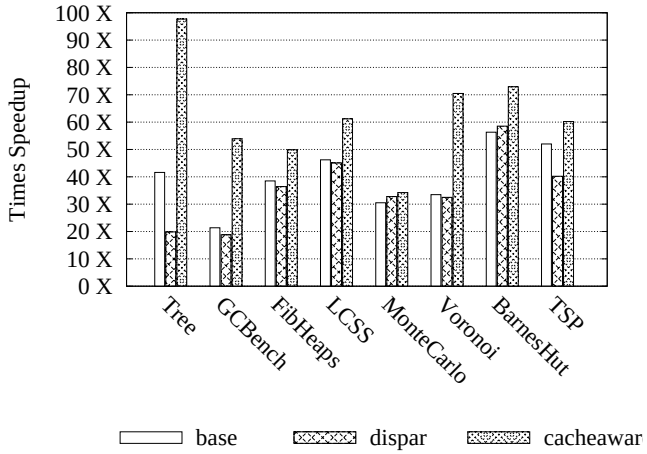

Figure 9: Mutator Speedup

Figure 10 breaks down how time is spent in the garbage collector. The time spent in the collector is approximately evenly split between marking objects, compacting the heap, and updating object references. Applications that manipulate a few, relatively large primitive arrays will spend more time in the compact phase as the marking phase and update phase become relatively inexpensive.

Figure 11 shows how much free space was left in the heap on average after collection. Figure 12 shows how much free space was left in the heap after each garbage collection round for each benchmark. These figures verify that we selected reasonable heap sizes relative to our workloads. We did not tune the heap size to hit an exact utilization ratio because we selected relatively large memory page sizes such that most heap partitions contain exactly one memory page to improve

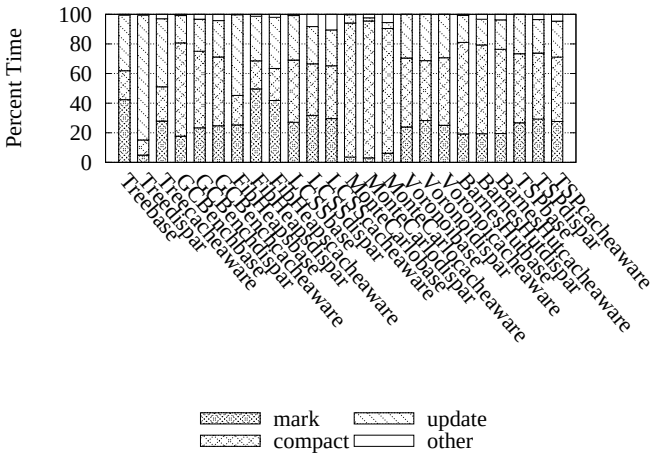

Figure 10: Breakdown of Time Spent in Collector

the TLB hit percentage and the processor only supports certain page sizes.

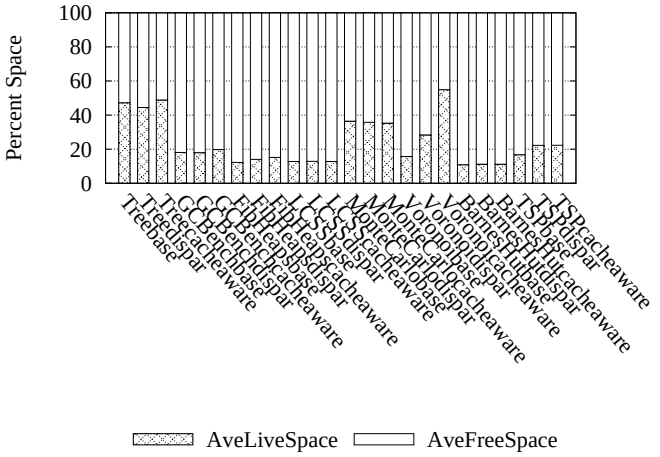

Figure 11: Average Free Space after Garbage Collection

6. RELATED WORK

The Multilisp collector was an early example of a parallel garbage collector [12]. Each processor has its own local semi-space heap that is collected using a copying collector. This approach is conceptually similar to the simple collector presented earlier in this paper and has the same problem with processors locally running out of memory when plenty of memory is globally available.

Flood et al. [11] employed dynamic per-object work stealing in their mark-compact copying collector and semi-space collector. Their approach fragments the heap into many pieces and cannot generate large contiguous blocks of free memory. Imai and Tick proposed a work stealing parallel copying collector [13]. Attanasio et al. [3] explored several parallel garbage collection algorithms including both generational and non-generational versions of copying and mark-and-sweep collectors. Cheng and Blelloch [8] devel-

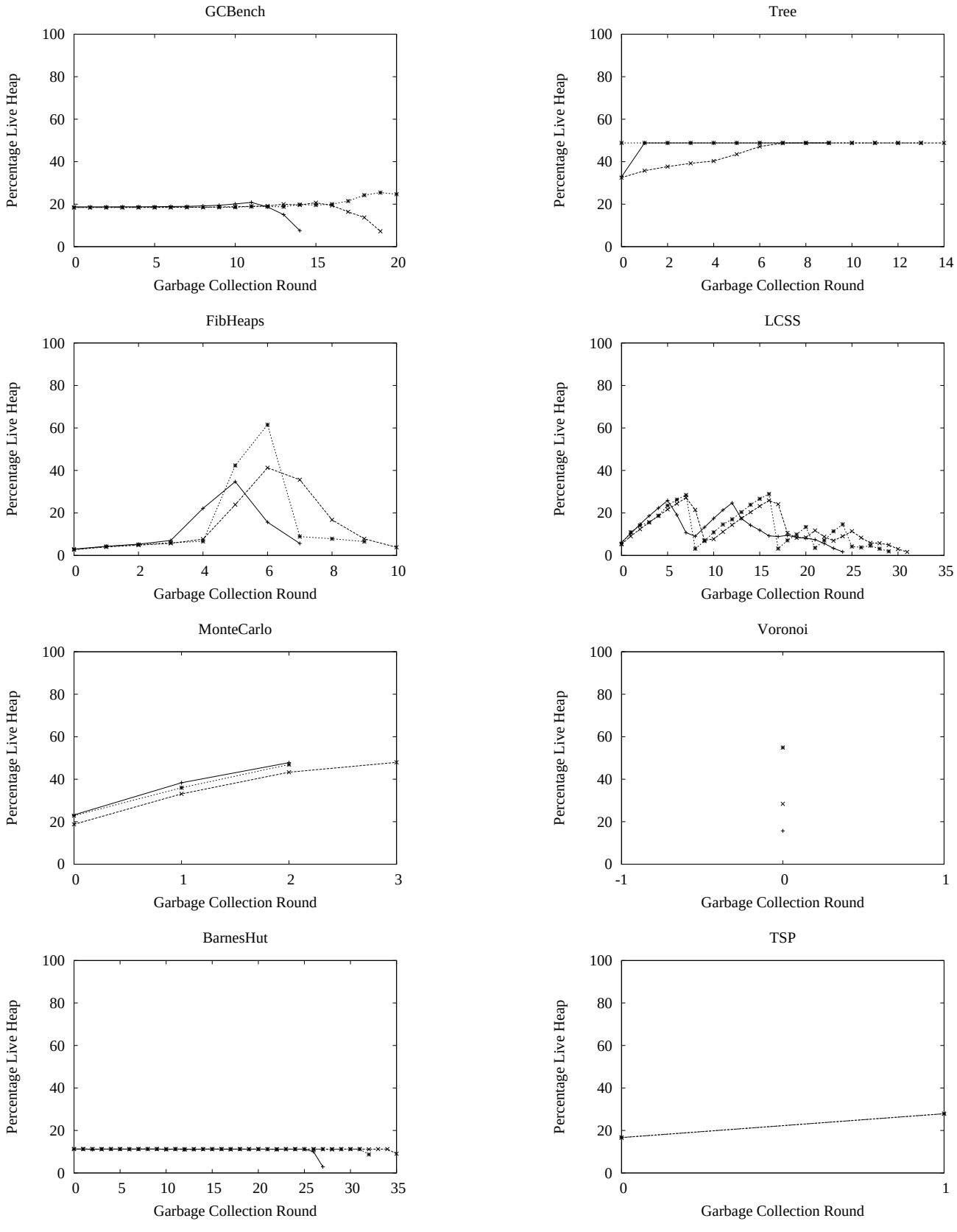

Figure 12: Free Space after Each Garbage Collection Round

oped a real-time GC, in which load balancing is achieved by employing a single shared stack among all threads. Ossia et al. [18] developed a parallel, incremental, and mostly concurrent garbage collector. Their load balancing mechanism called work packet management is similar to Imai's work pools. But their garbage collector partitions the global pool into sub-pools to reduce the atomic operations. All of these approaches do not address the memory locality concerns that have become important with recent processors.

A common theme in all of these collectors is load balancing between parallel threads. Many of these algorithms incur synchronization overheads to ensure load balancing. Our approach avoids the need for dynamic load balancing during collection (and its synchronization overhead) as our collection and allocation strategy inherently balances the work.

Endo et al. [10] developed a parallel mark-and-sweep collector based on work stealing. Their work does not address memory fragmentation.

Oancea et al. [17] presented a parallel tracing algorithm which associates the worklist to the memory space. Similarly to our approach, it also partitions the heap into regions. However, it does not statically map heap partitions to cores or processors. Instead, it binds worklists to heap partitions and lets the processors steal worklists. This work ignores fragmentation over the shared heap, which makes it difficult to support larger objects.

Cell GC [9] extends the Boehm-Demers-Weiser mark-sweep garbage collector to the Cell processor. It offloads the mark-phase to the synergistic co-processor so that the host processor can work on other computations.

Marlow et al present a block-based, parallel copying collector [16]. This collector copies objects first and then uses blocks to structure parallelization of scanning objects. This collector does not attempt to keep objects in the memory that is local to the allocating core and has to use separately allocated memory for objects larger than the block size. Immix is a region-based, parallel collector that uses defragmentation to defragment space within a region [4]. Unlike our collector, it is not designed to support objects that are larger than a region. Anderson explores the use of private nurseries to limit cache-coherence traffic over the bus [2]. They find that bus traffic become problematic with as few as 4 cores.

7. CONCLUSION

Garbage collectors will need to both manage locality and provide massively parallel collection to maximize performance on future many-core systems. We have implemented a garbage collector that balances data locality concerns with heap utilization and fragmentation concerns to achieve good performance while maintaining the abstraction of a single large heap. Our experience with several benchmarks indicates that the approach can achieve significant performance improvements due to both improved locality and parallelism. Our collector promises to provide the needed performance for future many-core processors, enabling applications in managed languages to scale to many-core processors.

8. REFERENCES

[1] nobench. `http://www.cs.york.ac.uk/fp/nobench/`, December 2007.

[2] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 21–30, 2010.

[3] C. Attanasio, D. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collector implementations, 2001.

[4] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and tutator performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2008.

[5] H. Boehm. Gcbench. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html`, 1997.

[6] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[8] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[9] C.-Y. Cher and M. Gschwind. Cell GC: Using the Cell synergistic processor as a garbage collection coprocessor. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 141–150, 2008.

[10] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–14, 1997.

[11] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[12] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[13] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, 1993.

[14] Single-chip cloud computer. `http://techresearch.intel.com/UserFiles/en-us/File/SCC_Sympossium_Mar162010_GML_final.pdf`, July 2010.

[15] X.-F. Li, L. Wang, and C. Yang. A fully parallel LISP2 compactor with preservation of the sliding properties. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[16] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory*

Management, pages 11–20, 2008.

[17] C. E. Oancea, A. Mycroft, and S. M. Watt. A new approach to parallelising tracing algorithms. In *Proceedings of the 2009 International Symposium on Memory Management*, pages 10–19, New York, NY, USA, 2009. ACM.

[18] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 129–140, New York, NY, USA, 2002. ACM.

[19] W. Partain. The nofib benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[20] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2008.

[21] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java Grande benchmark suite. In *Proceedings of SC2001*, 2001.

[22] Tilera. `http://www.tilera.com/`.