A Flexible, Efficient Concurrent Garbage Collector for Speculative Thread Processors

Michael Chen and Kunle Olukotun

Computer Systems Lab, Stanford University

Abstract

In this paper, we introduce a novel garbage collector for Java to be used for processors with speculative threads support like the Hydra chip multiprocessor (CMP). Thread speculation permits parallel execution of sections of sequential code with data dependencies enforced in the hardware, eliminating the need for explicit locking. We have augmented Dijkstra's classical on-the-fly mark and sweep collector to take advantage of the CMP's thread-level speculation and low-latency interprocessor communication. The resulting collector can execute concurrently without an explicit collector thread or can increase the collection rate by executing in parallel on all free processors. A dynamically scalable collector provides more control to adjust collection behavior according to the total system load and real-time deadlines. Speculative threads ensured the scalable implementation was simple and required no additional overheads, but we were unable to observe additional performance gains from eliminated explicit synchronization.

1. Introduction

The emergence of Java in the last several years has popularized the use of garbage collectors in real systems. Like many computing paradigms, garbage collection is subject to a fundamental cost-benefit trade-off. While garbage collectors simplify life for programmers, they have the potential to increase total memory usage, impose runtime overheads and exhibit long pauses that disrupt program execution. Most modern collectors fall into two categories, mark and sweep, and copying. Over the years, many variations on these two collector types have been proposed that use concurrency and incrementalization to minimize costs and undesirable collection pauses [20].

This paper presents a novel concurrent mark and sweep garbage collector for processors that support speculative threads, like the Hydra chip multiprocessor (CMP). Because memory ordering is enforced in the hardware, thread speculation permits parallel execution of sections of sequential code without explicit locking. We will show this helps make the collector implementation simple and flexible.

In our discussion, we first introduce the notion of thread-level speculation in Section 2. Sections 3 through 5 describe how the two programming models, procedural speculation and loop speculation, are used in our collector. We discuss the specific benefits and motivations behind our design, and show how our collector compares to other proposed concurrent mark and sweep implementations. Section 6 provides details from our experiments and shows how our implementation might realistically perform on our completed system. Finally, insights that we gained in our experiences designing the collector are described in Section 7.

2. Speculative Threads

2.1 Motivation

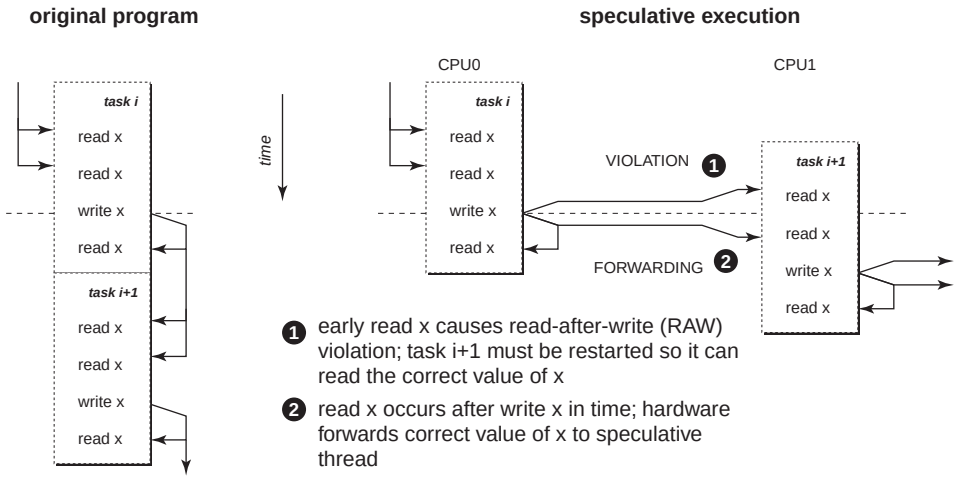

Figure 1 – Speculative execution of an arbitrary section of code

Memory speculation permits parallel execution of blocks of code that may contain unpredictable dependent memory references without explicit synchronization. These types of memory references are often found in integer-type applications and usually arise from modification of pointer references. The presence of such references makes these programs difficult to parallelize because expensive locks must be used or the program must be completely restructured.

Franklin and Sohi first proposed the basis of hardware memory speculation in the context of the Wisconsin Multiscalar project [32][21]. Their architecture is tailored to speculating on relatively fine-grained tasks. Our work targets the speculative thread model in the Hydra chip multiprocessor, which speculates on coarser-grained tasks [16][34]. In this system, a sequential program is divided into a ordered list of threads that may be run in parallel. To ensure the original sequential semantics hold, the hardware tracks all inter-thread memory dependencies, as illustrated in Figure 1. A write from a given speculative thread is forwarded only to later speculative threads that depend on the updated value according to the original memory reference ordering. If a later speculative thread in a sequence causes a memory data dependence violation by loading a value too early, the hardware ensures that this thread restarts and executes with the correct data.

Speculative threads are considerably different from synchronization mechanisms used to enforce dependencies on conventional shared-memory multiprocessors. Synchronization is normally inserted at compile time to guarantee a consistent ordering between read and writes to a given memory location [4]. During runtime, such synchronization can be computationally expensive and unnecessary. This is particularly true of integer code. In code that contains numerous pointer references and complex data structures, it is often the case that many potential dependencies may exist in a section of code, but a true dependency seldom occurs during execution of the program. Speculative threads can be less conservative than conventional parallel code in these situations. During runtime, we simply speculate on reads without synchronization and only back up and restart a thread if a read-after-write (RAW) violation occurs because of an actual dependency. A traditional multiprocessor, on the other hand, must be conservative and always synchronize at points when a dependency might occur.

2.2 The Hydra/KaffeVM System

In our current implementation, Hydra is comprised of four single-issue pipelined R4000 MIPS processor cores. Each processor has a private 16K L1 data and 16K L1 instruction cache, but share a large 2MB on-chip L2 cache. This organization permits low-latency communication between processors through the L2 cache that can be as little as 10 cycles [27]. We estimate that Hydra occupies the equivalent area of a 4-way superscalar processor in the same technology, with far less overall design complexity and higher possible clock rates [27].

For our Java virtual machine (JVM), we chose Kaffe [39] because it is one of the few well-supported JVMs ported to the MIPS platform for which source code can be readily obtained. Kaffe includes a just-in-time (JIT) compiler that performs simple register allocation, optimizes some common bytecodes, and includes GUI libraries for abstract-windowing-toolkit (AWT) and SwingSet applications. These features have allowed us to study the execution behavior of a wide range of Java applications on Hydra.

Hardware support for thread speculation only represents a small fraction of the total die area. Aside from controller overheads and complexity added to the base memory system, 2 bits plus 2 bits per word must be added to each L1 data cache line to track speculative read state. At least four write buffers, one associated per speculative thread, must also be available to hold uncommitted writes [15]. Currently, each buffer can hold up to 2KB of cache lines modified with speculative writes. We estimate that the area overheads these structures add to the system die area are equivalent to a pair of L1 caches.

2.3 Speculation on the Hydra CMP

	Procedure speculation	**Loop speculation**
Function	Execute procedure continuation/return speculative with predicted return value Execute procedure as head	Execute successive loop iterations speculatively Can apply `for`, `while`, and `until` loops Can handle mid-loop `break`, `goto`, and `continue`
Optimal usage	Procedures with significant and predictable amounts of work (e.g. leaf procedures) Less than 2KB of writes to different addresses Exact knowledge of which registers to pass to speculative processor Limited dependencies between caller and callee, preferable near procedures start to maximize overlap Predictable or void return values	Loop bodies with significant amount of work Less than 2KB of writes to different addresses Limited dependencies between iterations, preferably near loop iteration head to maximize overlap
Overheads (cycles)	procedure start 54 cycles procedure violation 12 cycles procedure end 24 cycles	loop start 22 cycles loop end of iteration 14 cycles loop violation 9 cycles loop terminate 41 cycles
Other	Finding good speculative procedures in programs is challenging	Need to isolate true dependencies on procedure local variables from false dependencies

Table 1 – Summary of procedural and loop speculation.

While code can be arbitrarily broken up into threads, we have limited decompositions on our machine to common programming paradigms to ease programmability, simplify code generation by the compiler, and minimize speculative overheads. The Hydra speculative hardware currently supports two programming models [15], described in Table 1. The models differ by how speculative threads are mapped to real processors and by their respective overheads.

The first speculative mode, known as procedural speculation, attempts to run a nested procedure call in parallel with the calling procedure (see Figure 2). At a procedure call that is to be executed speculatively, the procedure is executed as the "head," or non-speculative, thread, and the continuation, or the call return, is executed as a speculative thread. The most significant overheads come from passing live registers to the processor executing the procedure continuation speculatively. The runtime stack does not need to be explicitly passed to this processor because it is properly resolved by the speculation hardware. Stack values are passed on demand from the head processor to the processor executing the speculative thread through hits in the shared L2 cache in as little as 5 cycles per 32 byte line. For non-`void` functions, a return value prediction must also be generated for the speculative thread to use.

The second speculative mode, also shown in Figure 2, is restricted to loops. Here, several iterations of a loop body can be executed speculatively on the available processors. As long as the number of loop-carried dependencies is relatively small and well-aligned, the loop bodies can execute in parallel to achieve speedup. Our speculation model can easily handle the common loop based structures (e.g. `for`, `while`, `do... while`) as well as complex loops with multiple exit points (e.g. `break`, `continue`, `goto`). Targeting loops makes sense since many critical code sections are composed of compute-intensive loops. By optimizing speculative threads for loops on the Hydra CMP, we reduce the per iteration overheads of speculation, allowing us to execute iterations more quickly and apply loop speculation to smaller loop bodies.

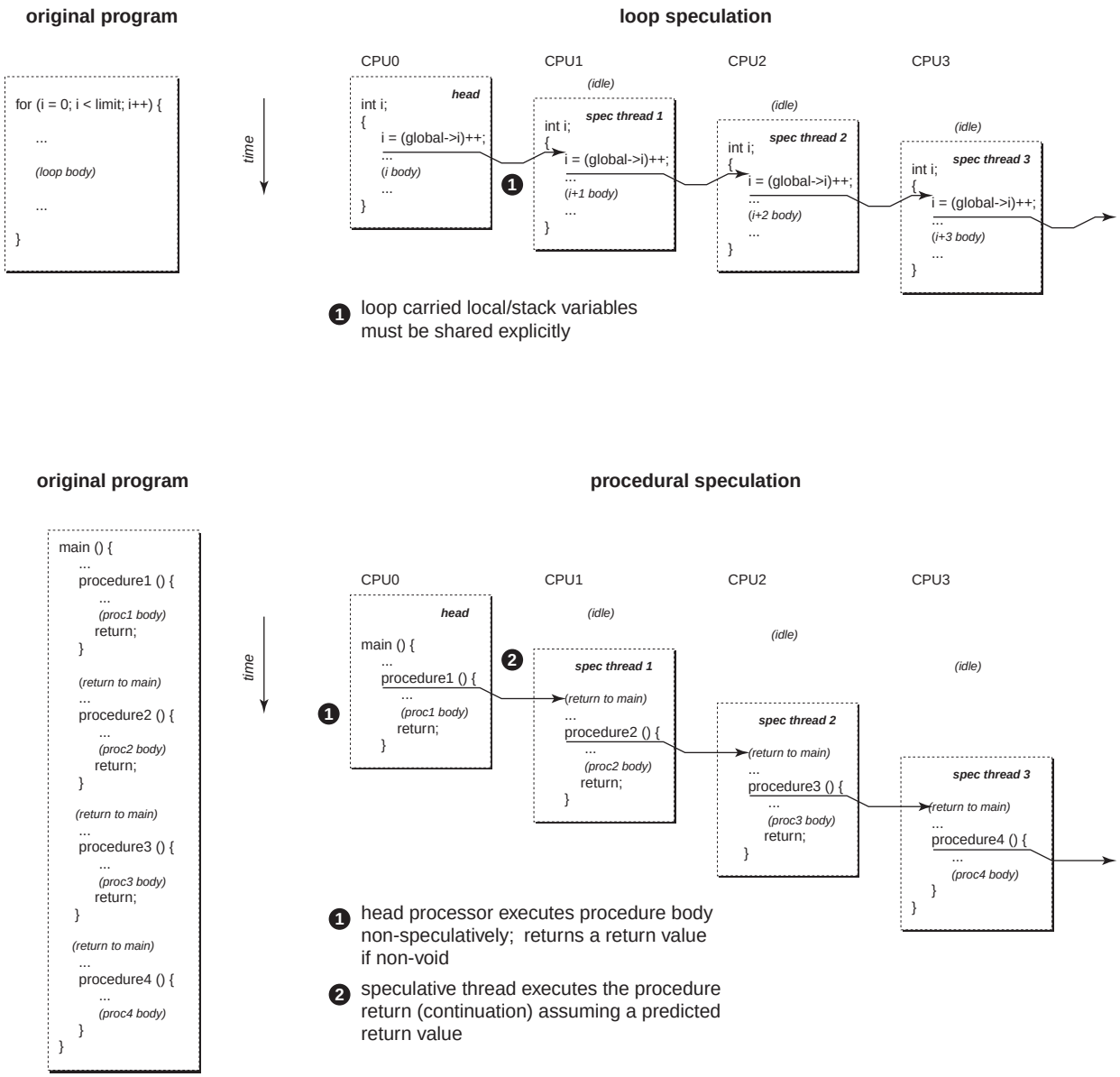

Figure 2 – Depiction of how procedural and loop speculation works.

Hydra has been shown to effectively speedup data parallel applications with reduced effort from the programmer and without any need to explicitly parallelize the source program [REF]. Getting significant speedups with integer code under speculation has been challenging because these programs tend not to have critical sections of code, and they have not been written with concurrency or parallelism in mind. In such instances, we have been able to improve speculative performance by eliminating false dependencies and by minimizing the effects of true dependencies. Moving dependencies within speculative threads to maximize overlap, or using speculative read locks that stall speculative threads rather than risk violation on a dependency can minimize the impact of true dependencies. An additional challenge specific to speculative procedures is locating calls that have predictable return values [23]. Detailed discussions on experiences with these techniques can be found in [3][28][29].

Another challenge is finding appropriately sized regions for speculative execution. Blocks of code to be executed as speculative threads must be large enough to speed up even with speculative overheads, but small enough not to overflow hardware buffers. As described in Section 2.2, speculative read state is stored in L1 cache lines and speculative writes are held in write buffers. Overflowing a write buffer or dropping an L1 cache line with speculative state stalls speculative execution. If this happens, a thread can only progress after it becomes the "head" thread and is no longer speculative, at which point it can safely execute reads and writes directly to memory. This effectively limits the size of a speculative thread. Of course, this upper bound is only limited by the buffer space available in our implementation. We can easily scale speculative execution to larger threads by increasing the size of hardware buffers in future implementations.

3. Base Mark and Sweep Collector

Our goal is to target the Hydra/KaffeVM system for embedded applications. In embedded environments, CPU cycles and memory must be used efficiently and predictably, and soft real-time behavior is generally desired. To satisfy these requirements, our implementation is based on Dijkstra's mark and sweep on-the-fly collector [7][8]. Mark and sweep makes the best use of available memory, maximally allowing the heap to be full of live objects. The basic mark and sweep collector assumes the mutator has been stopped, resulting in undesirable pauses. Dijkstra's algorithm extends the basic algorithm to allow collection to take place concurrently with mutator execution.

Dijkstra's algorithm achieves concurrency between mutator and collector by using the `shade()` operation. The `shade()` operation, given in Figure 3, acts as a write barrier and maintains the invariant that black objects never point to white objects. In our collector, this operation is inlined on all field and array writes to references, but not for writes to the runtime stack because it would be too expensive. Instead, references in the runtime stack are protected through conservative marking prior to sweeping.

```
shade(P) =
       if white(P)
              color(P) = gray
```

Figure 3 – Djikstra's `shade()` operator.

Mark and sweep collection is more appropriate than copying collection in an embedded environment. Compacting benefits of copying collection are less important here because paging and virtual memory are typically not used in these systems. Copying collectors are also cited for fast allocation behavior, but fast allocation is still possible using other techniques, as we will describe below. The need to finalize objects also complicates a copying collector implementation. The `finalize()` method, implemented by all Java objects and executed before a object is returned to the free heap, provides a chance to free up resources that cannot be freed automatically by the collector [Gosling et al 96]. To satisfy this requirement, a copying collector must scan the from-space at some point to identify the location of free objects so that the `finalize()` method can be executed on them. Finally, copying collectors produce significant write traffic on the memory bus proportional to surviving objects and halve the maximum live heap size because they require two half-spaces. Getting good real-time and incremental behavior from copying collectors also comes at a high price. Concurrent copying collectors require read barriers that can be considered more expensive than write barriers used by incremental mark and sweep collectors solely on the fact that heap reads generally occur an order of magnitude more often than writes [5][20]. Copying collectors that incrementalize collection through generations require complex pointer structures and tend to increase the number of CPU cycles devoted to collection [20].

Our baseline collector, without any use of speculative threads, features a fast allocator. Experiments and results collected by Dieckmann [5] on SPECjvm98 [35] applications suggest that Java objects can be reliably classified into two categories: short-lived, non-array objects less than 200 bytes that comprise the majority of the heap, and large objects and reference arrays that tend to have long heap lifetimes. These results led us to devise two object types in our heap for allocation, small pre-allocated objects (`objs`) and large, variable sized demand-allocated objects (`blobs`). The two object types are managed under separate systems for efficiency reasons, although they both receive memory from the same underlying page-based allocator. Non-array Java object sizes are known once the class and superclass layout has been examined. For small non-array objects, efficient allocation is possible by maintaining a pointer in the Class descriptor to a free list with objects of the appropriate size. Using this organization, allocation is as simple as removing an object from this free list. Allocation for large objects and arrays cannot take advantage of this optimization and, consequently, is considerably more expensive. Benchmark data suggests this is by far the less common case.

We should note some minor differences between our implementation and Djikstra's basic algorithm. We only `shade()` objects when the referring object is gray or black, while the basic algorithm simply shades on any pointer reference change. In this respect, our write barrier more closely resembles the one proposed by Steele [33]. While this write barrier is slightly more expensive, it is less conservative and reduces the amount of surviving garbage. To reduce the complexity of Djikstra's marking algorithm, which scales quadratically with the size of the heap, we use a linked list to represent gray objects instead of marking bits [20].

The collector also has other features and optimizations. The memory allocator and garbage collector can operate safely with multiple mutator threads. The allocator can allocate to multiple mutators concurrently through a per-page locking scheme. Djikstra's base algorithm only permits incremental marking, but must stop the mutator during sweeping. To facilitate sweeping without stopping the mutator, new objects being allocated by the mutator are separated from objects that may be freed during concurrent sweeping [20]. Since references constitute about 50% of per object data [5], we implemented exact collection to minimize the time required to walk each object. We also eliminated finalization calls for objects whose `finalize()` method simply maps to an empty method in the Object superclass.

4. Garbage Collection with Speculative Threads

4.1 Parallelizing garbage collection

Table 2 shows the work performed by a single iteration of the mark and sweep collector. We have decomposed collection into regions that constitute independent critical phases. Phases {2} to {6} are well-defined loops that may benefit from speculative threads. Phases {2} and {4} are identical, but must be performed twice, since phase {3} may add new objects to the gray list. For each critical phase, we will discuss issues to consider, and how each can be parallelized using speculative threads and traditional parallelization.

Number	Phase	Function
{1}	*Mark root objects*	Look for root objects (thread stack, default class loader, root references) and place on gray list.
{2}*	*Mark gray objects black*	Walk marking list and blacken scanned objects until list is empty.
{3}*	*Identify objects to finalize*	Scan white objects to find any objects that need to be finalized. Mark dependent objects of objects to be finalized, in case they get reattached during finalization.
{4}*	*Mark gray objects black*	Walk marking list and blacken scanned objects until list is empty.
{5}*	*Sweep white objects*	Return white objects which do not need finalization to allocator.
{6}*	*Finalize objects*	Invoke `finalize()` method on objects and free.

Table 2 – A typical garbage collection cycle (asterisk denotes critical phases we studied).

In terms of CPU time, marking objects (phases {2} and {4}) constitute the largest fraction of the collection cycle. To mark an object, we must first examine all its descendants. Descendants that are white must be shaded to gray and added to the marking list. Once an object's descendants are either gray or black, the object can be shaded black. This process can be parallelized using speculative threads or by traditional means.

In a traditional parallel implementation of the marking phase, locks must be placed around structures that might be shared by more than one process. In the simplest implementation, accesses to the marking list must be synchronized between processes removing objects to mark and adding new objects to be marked. Additionally, the object being examined must also be locked to prevent marking by another process. A more robust implementation, suggested by Endo [11], deals with potential lock contention by using private marking queues. Significant complexity is added, though, because a stealing algorithm is required to help balance load between processors.

Our speculative thread implementation uses loop speculation, and does not require explicit locking because data dependencies are detected and resolved by the hardware. All speculative threads get objects to process from a single shared marking list. Like Endo [11], though, we maintain a private list in each thread to eliminate dependency violations from new objects being added to a shared marking list. After the global marking list has been emptied, the speculative threads merge gray objects from their respective private lists back into the global list and then return to marking objects. This process is repeated until the marking list is empty.

After marking all the live objects, garbage objects for finalization must be located by phase {3}. Objects cannot be freed immediately. Descendants of an object to be finalized must be marked because they may be reattached to live memory during that object's finalization. Since more objects may be added to the marking list after objects to finalize have been located, we must also re-execute object marking as phase {4}.

Our speculative implementation of phase {3} reflects the fact that identifying objects to finalize bears some resemblance to the marking phase. For both speculative thread and traditional parallel implementations, private marking lists still eliminate unnecessary dependencies. A traditional parallel implementation must continue to lock objects being examined to prevent concurrency problems. A shared job queue, though, can be eliminated in phase {3} for both implementations. Because set of memory we must examine to locate objects to finalize is initially fixed, we can partition the scan into tasks that avoid contention.

Phases {5} and {6} cleanup memory after reachable objects have been marked. In phase {5}, the collector sweeps live memory for objects to be freed. Objects that do not need to be finalized or have a finalize method that does no work can be freed immediately, otherwise they are forwarded to the finalizer (phase {6}). The finalizer executes the `finalize()` method for objects that need finalization. Phases {5} and {6} may be merged into a single loop, although this is not the case in our current implementation.

As before, we can eliminate a shared job queue in a concurrent implementation through intelligent partitioning of memory. With a traditionally parallelized implementation, though, we have to be careful about concurrent accesses from the sweeper and finalizer back into the virtual machine. For example, there must be some well-designed synchronization back into the allocator as garbage objects are returned to free lists that can protect internal VM structures without limiting concurrency. The speculative thread collector does not need to worry about these issues and can forgo locking within the VM since our speculation model guarantees ordering between the threads.

4.2 Eliminating the concurrent collection thread

Procedural speculation can facilitate concurrent collection without running a separate, explicit collection process concurrently with the mutator. As previously described, in Dijkstra's on-the-fly implementation, the black to white invariant is maintained between the collector and mutator through the `shade()` operation. We can use this `shade()` operation to do some additional useful work for the collector, executing any of the work described in Table 2.

This idea has been proposed previously as a means to incrementalize mark and sweep collection and does have certain advantages [20]. In particular, it eliminates any need to synchronize between the mutator and a separate collector process. Normally, the incrementalized collector work executes within the same process as the mutator. Unfortunately, this slows down the mutator's progress.

Procedural speculation allows us to execute this code in parallel with the mutator without a concurrent collector process. Since the `shade()` operator does not return a value, we do not need to be concerned about speculative violations caused by return value mispredictions. The amount of incremental work performed in the `shade()` function call can be chosen accordingly so that there will be enough work to amortize speculative overheads, but not so much as to overflow speculative hardware buffers. Finally, we do not expect violations between the mutator and collector since they primarily work on different fields within a given object. Only the collector reads and writes the collection fields in objects (e.g. object color). Assuming one level of thread speculation, the mutator executing the continuation from the `shade()` call is speculative relative to collection work with this call. Consequentially, the `shade()` call can not cause violations to the mutator executing on the speculative thread because the collector does not modify live object fields that the mutator accesses.

Speculative violations would primarily results from violations caused by closely spaced speculative `shade()` calls, as might occur when two field or arrays are written within the span of a few instructions. If a violation does occur in this situation, the speculation hardware would cause the second `shade()` call to abort and restart. At no point in time is correctness compromised.

4.3 Using the speculative garbage collector

	Speculation	**Collector threads**	**Notes**
Minimal collection	Procedural speculation of `shade()` write barrier	0	Minimal collection without separate collector thread running. Work must be bounded to prevent stalls.
Normal collection	None	1	Mutator `shade()` operator forwards incremental updates to collector.

			Collector thread runs concurrently with mutator threads.
Maximum collection	Loop speculation of collector phases	2-4	Increased collection rate can be used to meet deadlines. Can be run concurrently with mutator or when mutator is stopped.

Table 3 – Possible collector modes.

The collector we have designed can be run in various modes. With the speculative `shade()` function and speculative loop collection routines, the collector can be scaled to run on zero to four processors, as outlined in Table 3. In minimal collection mode, the collector thread is disabled, and the `shade()` operation in the mutator is run speculatively, as described in Section 4.2. Normal collection is simply Djikstra's classical organization with one collector thread executing concurrently with the mutator. In this mode, `shade()` operations by the mutator are not speculative, and simply forward objects to the collector thread. To increase the rate of collection, parallel collection discussed in Section 4.1 can be executed concurrently with the mutator thread. Finally, mutator progress can be halted and incremental collection of up to four processors can be started for maximal collection rates.

The scalability of the collector allows it to easily adapt to soft real-time constraints and system load on the Hydra CMP without any additional overheads to the system. This allows us to address important concerns of garbage collection in embedded systems. Scalable collection behavior makes it easier for the collector to meet soft real-time deadlines and get better incremental behavior. If collection is not keeping up with the mutator memory allocation rate, collection rates can be dynamically increased by dedicating more processors to collection. When application code requires more CPU cycles, like when a multithreaded application is utilizing all the available processors, processors allocated to collection can be minimized.

Implementing a similar collector using traditional parallelization is possible, but would require more system resources and more careful programming. At a minimum, multiple collector threads must be created at the operating system level. This introduces creation, context switching, and process control overheads at the system level. Appropriate locks are required to ensure proper synchronization and correctness. More complex data structures must be introduced to minimize concurrency bottlenecks, handle load balancing between threads, and accommodate changing number of processors allocated to collection.

Similar overheads for speculative thread creation and context switching are smaller. Because low-level handlers nested beneath the operating system manage speculative threads, they are more inexpensive to create and manage. Speculative threads are also easier to develop for. The hardware ensures a consistent thread ordering so that programmer-inserted synchronization is unnecessary and concurrent execution is always guaranteed to be safe and correct.

5. Comparisons to Other Concurrent Collectors

Apart from Djikstra's classical algorithm, other proposals have been proposed to increase concurrency in mark and sweep collectors. Huelsbergen [19] and Queinnec [29] suggest changes to the basic algorithm to expose very coarse parallelism by overlapping the mark and sweep phases of collection. Both of these do well to pipeline the mark and sweep sweep phases, but do not expose collection parallelism scalable to more than three processors. Additionally, Huelsbergen [19] suffers from global barrier synchronization between the mutator, marker and sweeper after each collection, and Queinnec [29] minimally requires barrier synchronization between the mark and sweep threads per garbage collection cycle.

[11] discuss their implementation of parallel mark and sweep on a traditional multiprocessor. While they discuss some issues relevant to our design, their paper focuses primarily on large scale multiprocessors (>16 processors). Large scale multiprocessors present different problems including larger communication overheads for shared variables and locks, and problems with contention between large number of processors. They also base their work around a stop and collect implementation that does not address incrementalization.

There have also been papers that describe concurrent collection in the presence of multiple mutators by Doligez [9][10]. These papers suggest changes to Djikstra's algorithm to accommodate multiple mutators and to eliminate the need for an atomic `shade()` operator. Doligez [10] describes fixes for hypothetical collection races for unsynchronized read/write accesses by two mutators to a shared object, but we feel these changes to Djikstra's base algorithm are unnecessary for our Java VM environment. Fixing these races is not useful considering fundamental concurrency problems that arise when accessing an unsynchronized object shared between two threads. The proposed modifications to the base algorithm to eliminate the need for an atomic `shade()` also add conservatism, causing garbage to survive for at least two collection cycles.

Performance Evaluation

To estimate the performance of our collector, we studied the effectiveness of the two primary components of our implementation, the collection routines based on loop speculation discussed in Section 4.1, and incremental `shade()` operation executing under procedural speculation.

A cycle accurate simulator of the Hydra CMP, `LESS`, was used to collect detailed performance numbers. `LESS` executes real MIPS code with appropriate instruction and memory latencies. The simulator also models the behavior of the speculation hardware for applications that take advantage it. We performed comparative analyses between baseline and modified versions of the garbage collector. Unfortunately, `LESS` executes about 3k simulated instructions per processor per second, which makes it difficult to perform complete simulations of benchmark programs. Simulating critical phases of small collection cycles, containing 5k-10k objects typically takes half an hour. Procedural speculation is even more time consuming since we usually have to simulate significant amounts of application code before encountering a speculative `shade()` operation.

As a result, we found it more useful to sample critical phases of the garbage collection cycle to get performance figures for speculative loops. For speculative procedures, we only looked at behavior in sections with high frequencies of `shade()` operations. Because we found little variation in performance between samples in different programs and different sections within the same program, we believe this method produced adequate data to study the behavior of our system.

These performance results were supplemented with number showing typical collector usage on actual applications. Statistics like average marked objects per `shade()` operation help us understand how we can apply the speculative collector in a real system, as we will describe in the next section. Using simple timing macros inserted into the Java virtual machine, we collected numbers from two real systems, a SGI IRIX machine with a 200MHz MIPS R4000 processor and a Linux Intel with a 333MHz Celeron™ processor.

Name	Binary size (KB)	Allocation (MB)	Description
SPECjvm98 – compress	18	334	LZW compression
SPECjvm98 – db	12	224	Database functions on memory resident database
SPECjvm98 – jack	131	481	Java parser generation based on PCCTS
SPECjvm98 – javac	561	518	Java compiler from JDK 1.0.2
SPECjvm98 – jess	386	784	Implementation of NASA's CLIPS expert shell system
SPECjvm98 – mtrt	56	355	Raytrace of dinosaur using two threads
SwingSet	286	104	Widget demo of Java's advanced GUI included in JDK1.2+
jBYTEmark	71	11	BYTE magazine's benchmark suite of 10 small applications

Table 4 – Benchmark descriptions and characteristics.

For benchmarking, we needed Java applications that sufficiently exercise the garbage collector. Descriptions of the applications that we studied are given in Table 4. The SPECjvm98 benchmark suite is an excellent source of realistic programs that are also allocation intensive. We also included SwingSet, a demo of Java's graphical user interface (GUI) environment, to capture typical garbage collection behavior for user-interactive applications. The various statistics on collection behavior of the benchmark applications are summarized in Table 5.

Benchmark	Collection Cycles	Marked Objs / Collection Cycle	Allocated Objs / Collection Cycle	Freed Objs / Collection Cycle	Finalized Objs / Collection Cycle	shade() calls / Collection Cycle	Marked Obj / shade() call	Allocated Obj / shade() call	Freed Obj / shade() call	Finalized Obj / shade() call
compress	20	3257	255	70	13	25.2	129	10	3	1
db	100	291042	60712	57736	39	11.9	24396	5089	4840	3
jack	86	40482	195474	173264	21315	2056.6	20	95	84	10
javac	32	164156	90003	66776	893	5837.4	28	15	11	0
jess	72	25437	109722	109251	88	195.2	130	562	560	0
mtrt	36	114641	177406	176302	1002	14.4	7937	12282	12206	69
SwingSet	10	33443	24510	12058	1312	1783.8	19	14	7	1
jBTYEmark	96	1790	1299	1172	109	87.0	21	15	13	1

Table 5 – Garbage collection statistics for our benchmark programs.

A quick scan of numbers in Table 5 reveals that the collection characteristics vary greatly between the selected programs. In particular, we found collection overheads to be smaller than expected in the Swing environment, probably due to code that uses memory effectively so the garbage collector is invoked less frequently. Across the benchmarks, collection generally constituted from < 1% to over 20% of the total execution time on the KaffeVM (results not shown for brevity). As we improve the quality of the native code from the Kaffe JIT compiler, we expect more time will be spent in collection. We have performed relative comparisons between the best JIT compilers that show Sun's HotSpot or IBM's JIT to be up to 2x faster than Kaffe's, suggesting collection can approach 1/3 of the total execution time on some applications if we improve the JIT compiled code.

6. Performance Results

Collector region	Implementation	Average iteration size (cycles)	Objects / iteration	Speedup
Mark objects {2}{4}	Procedure speculation		10-200	
	Loop speculation	2000	4	2.41
	Traditional parallel	381	1	2.61
Identify objects to finalize {3}	Procedure speculation		40-800	
	Loop speculation	735	5	2.22

	Traditional parallel	164	1	2.20
Sweep objects {5}	Procedure speculation		50-1000	
	Loop speculation	647	5	2.09
	Traditional parallel	172	1	2.11
Finalize objects {6}	Procedure speculation		10-200	
	Loop speculation	750	1	3.41
	Traditional parallel	900	1	3.1

Table 6 – Performance improvements using speculative loops.

6.1 Parallelized collection using loop speculation

To investigate the effectiveness of loop speculation to parallelize collection, we implemented three different versions of each critical phase given in Table 2. A serial version was used as a baseline to compare a version that used speculative loops and one that was parallelized using traditional methods. In this section, we describe our implementation experiences and the results from simulation.

Our implementations are based on the observations described in Section 4. Reasonable attempts were made to eliminate bottlenecks due to shared variables and to only use synchronization when necessary. As described earlier, the speculative version of the critical loops does not require synchronization where the traditionally parallelized implementation does, since concurrent correctness is guaranteed by the speculative hardware. We were able to eliminate a source of contention in the traditionally parallelized implementation and a source of violations in the speculative version of phases {3} and {5} through intelligent partitioning of the memory to be scanned.

Overall, developing with speculation simplified programming because correctness is guaranteed relative to a sequential version of the program. As a result, we were able to focus our attention completely on performance tuning. Solutions and approaches for tuning applications under speculation differ somewhat from traditional methods. With speculation, most of the work involves optimizing true and false dependencies between speculative threads. Violation statistics recorded by our simulator were used to locate and deal appropriately with each of these dependencies. From our experience, the most relevant pieces of information for performance debugging are the store instruction PC, the violating address, and relative time of the violation within a speculative task. The actual hardware will be capable of collecting similar information for debugging purposes.

Results from our experiments, shown in Table 7, suggest the two parallel versions yield similar performance. Initially, we hoped that the speculative loops would be faster because they don't suffer from locking overheads. We have two explanations for the observed behavior. First, both versions benefit from the low-latency interprocess communication of the chip multiprocessor that significantly reduces the communication cost of shared locks that protected shared variables. Consequentially, the overheads seen by the traditionally parallelized version are not much greater relative to the version that relies on the speculative hardware to properly synchronize access to shared variables.

Secondly, the speculative loop performance may be limited by variations or imbalance in work performed by each iteration. This behavior is apparent in the marking phase where individual objects are of different sizes and contain varying numbers of reference fields. We can easily expect great variation in the number of descendants that must be examined and marked for different objects. Because speculative threads obey a sequential ordering, speculative loops iteration may be stuck behind a slow iteration that has more work to do. Such stalls force processors to idle and can limit speedup.

To amortize per-iteration speculative overheads and to get better average processing time per iteration, we process multiple objects per speculative loop iteration. To study the effectiveness, we implemented the idea and then observed the speedup as a function of the grouping size, shown in Table 7. Limited grouping modestly improved the performance of phases {2}{4} and phase {5}. Phase {6} did not benefit from grouping. Phase {3} appears to benefit significantly from grouping, but it is difficult to ascertain. Our observations indicate phase {3} generally has long stretches of tight loops (scanning memory) and bursts of heavy work per iteration (marking objects to finalize), resulting in high per iteration variance in work. Consequently, speedups with grouping for phase {3} can increase significantly with grouping during memory scanning, but quickly deteriorate if objects to mark are clustered together.

	Speedup by objects / iteration						**Loop size by objects / iteration** (cycles)					
Collector region	**1**	**2**	**4**	**6**	**8**	**10**	**1**	**2**	**4**	**6**	**8**	**10**
Mark objects {2}{4}	2.18	2.16	2.21	2.22	2.12		696	1340	2590	3818	5200	
Identify final objects {3}	1.51	1.91	2.21	2.30	2.34	2.37	173	325	639	959	1273	1593
Sweep objects {5}	1.79	2.03	2.06	1.98	1.89	1.83	156	285	515	726	914	1125
Finalize objects {6}	3.35	1.19					810	1495				

Table 7 – Garbage collection statistics for our benchmark programs.

Marking objects, phases {2} and {4}, and object finalization, phase {6}, generates the most significant speedups, between 2.5 and 3.5. We are only able to get speedups of slightly over 2 for the other two phases, object sweeping (phase {5}) and scanning for object finalization (phase {3}). We believe that speedup in these two phases is probably limited by bandwidth to main memory. Objects are not likely to be found in the L1 or L2 cache because objects are only scanned once and in an unpredictable reference pattern. With independent object accesses from four processors to main memory, we suspect the processors are being stalled by limited main memory bandwidth. In the two later critical phases, less work is performed per object scanned, so a larger fraction of time is being spent accessing objects from main memory. Thus, while shared variable communication overheads between processors are minimized in our configuration, the limited total bandwidth to main memory shared between four processors hurts our performance on memory intensive code.

6.2 Concurrent collection using procedural speculation

The results observed from using procedural speculation on the `shade()` operator in the mutator are encouraging. With appropriately sized incremental collector work assigned within the call, we are able to completely hide collection overheads from mutator progress without a separate collector thread. To help understand the limitations of this approach, we experimented with sections of application code that invoked the `shade()` operator. We varied the amount of incremental collector work performed within the `shade()` operator.

These experiments reveal that procedural speculation should only be used as a function of the amount of incremental collector work available. When one invocation of the `shade()` operator is assigned only a small amount of work, like scanning a small number of objects, procedural speculation is not effective. In this case, the overheads of procedure speculation are greater than the work performed within the call, resulting in a slowdown rather than a speedup.

Speedup can also be limited if the `shade()` operator tries to complete a large amount of incremental work. This happens because the hardware can only buffer a finite record of speculative state. As stated earlier, this state is used to detect ordering violations and undo writes when dependency violations occur. According to sequential ordering, reads and writes after the continuation, or return to the mutator, are considered speculative relative to the `shade()` operator. So the speculative state that needs to be stored is reads and writes performed by the mutator. If the speculative buffers fill completely, then the mutator must stall until the `shade()` operation completes and the mutator is no longer speculative. Consequently, it is difficult to place an upper bound on the amount of work that should be assigned to a speculative `shade()` operation because limits are dictated by how the mutator application uses memory.

Table 7 summarizes what our experiments revealed to be ideal conditions to use procedural speculation. Upper limits on work assigned to speculative `shade()` are based on observations from our experiments. Because these limits are application dependent, we are still investigating ways of calculating an accurate metric. According to Table 7, these limits fall within the average objects than need to be examined per `shade()` operator for many of the benchmark applications. These results suggest great flexibility in using the speculative `shade()` operator on real applications. In cases were more work must be performed, speculative execution of the `shade()` operator can be disabled and one of the other modes discussed in Section 4.3 can be used.

Future Work

Future research on the speculative garbage collector will focus on issues relating to the actual use of the collector in a real system. Our current idea is to develop a feedback based system for controlling collection. Such a system would incorporate into the virtual machine functions that take factors like system load, allocation and collection rate, and any soft real-time deadlines to control collector progress relative to the mutator.

The garbage collector system is just a single component of the final system we are developing. The speculative garbage collector will co-exist with speculation used in the application threads [3]. Dynamic runtime support in the Java virtual machine will mange the processors and the use of speculation. This system would balance the use of the speculative threads with Java threads. Multi-threaded applications can take advantage of the available processors and share data through low latency communication. Speculative support would provide another means to parallelize sequential applications that would be difficult to represent through explicit threads either because of fine-grained sharing or because of non-regular pointer structures and accesses. The ultimate goal is to keep processor utilization and performance high whether one program thread or many program threads are running.

7. Conclusions

This paper shows how to implement a very flexible mark and sweep collector using speculative threads. Collection can be dynamically scaled from zero to four processors under our system, facilitating better load balancing and adaptation to soft real-time constraints. In minimal collection mode, the collector thread is disabled, and the `shade()` operation in the mutator is run as a speculative procedure. To increase the rate of collection, parallel collection using speculative loops of up to four processors can be started. We primarily focus on the design and performance of the collector here, but future studies will focus on techniques to manage scalable collection behavior in a real system.

A speculative collector has certain advantages over a similar implementation that uses only traditional parallelization techniques. With speculation, scalable collection can be implemented cheaply, with minimal complexity added to the core collector and without additional overheads. Using speculative threads requires less memory and process management overheads, and guarantees correctness without explicit synchronization. The simplified programming model simplifies implementation, allowing us to focus on performance tuning.

We describe general issues to consider when designing parallel and concurrent collectors, including elimination of non-essential dependencies and minimization of shared variables. We show how these issues and speculation constraints like task size and overheads impacted the collector design. The effectiveness of the speculative implementation was evaluated using simulations. We show significant speedups in our speculative loop implementations of collector phases. We also demonstrate how speculative procedures can effectively perform concurrent collection without a separate collector thread.

In the comparative studies between critical collector phases, we find the speculative loop and traditional parallel implementations to perform similarly. We had hoped to demonstrate improved performance with speculation in these phases. We believe that improvements were not observed because the traditional implementation benefits from Hydra's low-latency interprocessor communication while speculative threads are limited inherently by their implicit ordering.

Acknowledgements

We thank Tim Wilkinson and Peter Mehlitz at Transvirtual Technologies (http://www.transvirtual.com) for the excellent support they have provided on the KaffeVM and their input into the design of this collector. Lance Hammond, author of the `LESS` simulator, was an invaluable source of help on debugging our simulations. US Defense Advanced Research Projects Agency contracts DABT 63-95-C-0089 and MDA 904-98-C-A933 provided funding for this research.

References

[1] J. Boyle et al., *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, 1987.

[2] H. Boehm, A. J. Demers and S. Shenker, "Mostly Parallel Garbage Collection," In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 157-164, 1991.

[3] M. K. Chen and K. Olukotun, "Exploiting Method-Level Parallelism in Single-Threaded Java Programs," In *Proceedings of Parallel Architecture and Compiler Technology (PACT'98)*, Paris, France, October, 1998.

[4] D. Culler, J.P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, CA 1998.

[5] S. Dieckmann and U. Holzle, "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks," UC Santa Barbara Technical Report, TRCS98-33, December 1998.

[6] A. Diwan, D. Tarditi and E. Moss, "Memory Subsystem Performance of Programs Using Copying Garbage Collection," In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages (PoPL-21)*, Portland, OR, January 1994.

[7] E. W. Dijkstra et al., "On-the-fly garbage collection: An exercise in cooperation," In *Lecture Notes in Computer Science, No. 46.* Springer-Verlag, New York, 1976.

[8] E. W. Dijkstra et al., "On-the-fly garbage collection: An exercise in cooperation," In *Communications of the ACM, 21(11):965-975*, November 1978.

[9] D. Doligez and X. Leroy, "A concurrent, generational garbage collector for a multithreaded implementation of ML," In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages (PoPL-20)*, Charleston, SC, 1993.

[10] D. Doligez and G. Gonthier, "Portable, Unobtrusive Garbage Collection for Multiprocessor Systems," In *Proceedings of the 21th Annual Symposium on Principles of Programming Languages (PoPL-21)*, Portland, OR, January 1994.

[11] T. Endo, K. Taura and A. Yonezawa, "A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines," In *Proceedings of High Performance Networking and Computing (SC97)*, San Jose, CA, November 1997.

[12] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," In *Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA-19)*, pages 56-67, Gold Coast, Australia, May 1992.

[13] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wessley, Reading, MA, 1996.

[14] R. H. Halstead, "Multilistp: A language for concurrent symbolic computation," In *ACM Transaction on Programming Languages and Systems*, 7(3):501-538, July 1985

[15] L. Hammond, M. Willey and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, September 1998.

[16] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen and K. Olukotun, "The Stanford Hydra Chip Multiprocessor," In *IEEE MICRO*, March – April 2000.

[17] W. Hennessey, "Real-Time Garbage Collection in a Multimedia Programming Language," In *OOPSLA 1993 Workshop on Memory Management and Garbage Collection*, September 19, 1993.

[18] U. Hölzle et al., "Java On Steroids: Sun's High-Performance Java Implementation," In *Hot Chips IX*, Sanford, CA, August 25-26 1997.

[19] L. Huelsbergen and P. Winterbottom, "Very Concurrent Mark & Sweep Garbage Collection without Fine-Grain Synchronization," In *1998 International Symposium on Memory Management*, 1998.

[20] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, Chichester, UK, 1996.

[21] T. Knight, “An Architecture for Mostly Functional Languages,” In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 500-519, August 1996.

[22] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*. Addison-Wessley, Reading, MA, 1997.

[23] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October, 1996.

[24] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “Dynamic Speculation and Synchronization of Data Dependences,” In *Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA-24)*, Denver, CO, 1997.

[25] S. Nettles and J. O’Toole, “Real-Time Replication Garbage Collection,” In *Proceedings of the ACM SIGPLAN ’93 Conference on Programming Languages Design and Implementation (PLDI’93)*, Albuquerque, NM, June 23-25 1993.

[26] S. Nettles and J. O’Toole, “Concurrent Replication Garbage Collection: An Implementation Report,” Carnegie Mellon University Technical Report, CMU-CS-93-138, April 1993.

[27] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Y. Chang, “The Case for a Single-Chip Multiprocessor,” In *Proceedings of the 7th International Symposium on Architectural Support for Parallel Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996.

[28] K. Olukotun, L. Hammond and M. Willey, “Improving the Performance of Speculatively Parallel Applications on the Hydra CMP,” In *Proceedings of 1999 ACM International Conference on Supercomputing (SC’99)*, Rhodes, Greece, June 1999.

[29] J. T. Oplinger, D. L. Heine, and M. S. Lam, “In Search of Speculative Thread-Level Parallelism,” *Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques (PACT’99)*, Newport Beach, CA, October 1999.

[30] C. Queinnec, B. Beaudoing, and J. P. Queille, “Mark DURING Sweep rather than Mark THEN Sweep,” In *PARLE ’89: Parallel Architectures and Languages Europe*, Eindhoven, Netherlands, June 1989.

[31] W. J. Schmidt and K. D. Nilsen, “Performance of a Hardware-Assisted Real-Time Garbage Collector,” In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[32] G. Sohi, S. Breach, and T. N. Vijaykumar, “Multiscalar Processors,” In *Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, Ligure, Italy, June 1995.

[33] G. L. Steele, “Multiprocessing compactifying garbage collection,” In *Communications of the ACM*, 18(9):495-508, September 1975.

[34] G. Steffan and T. C. Mowry, “The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization,” In 4th International Symposium on High Performance Computer Architecture *(HPCA-4)*, Las Vegas, NV, February 1998.

[35] Standard Performance Evaluation Corportation, *SPECjvm98 v1.00*, Manassas, VA, July 1998.

[36] D. Tarditi and A. Diwan, “The Full Cost of a Generational Copying Copying Garbage Collection Implementation,” In *OOPSLA’93 Workshop on Memory Management and Garbage Collection*, Washington, DC, September 1993.

[37] P. R. Wilson and M. S. Johnstone, “Real-Time Non-Copying Garbage Collection,” In *OOPSLA’93 Workshop on Memory Management and Garbage Collection*, Washington, DC, September 1993.

[38] S. Wholey and S. E. Fahlman, “The design of an instruction set for Common Lisp,” In Steele [LFP1984], pages 150-158, 1994.

[39] T. Wilkinson et al., *Kaffe Virtual Machine*, http://kaffe.org, 1997-2002.

[40] Benjamin Zorn, “Barrier methods for garbage collection,” University of Colorado Technical Report CU-CS-494-90, University of Colorado, Boulder, CO, November 1990.