The Pauseless GC Algorithm

Cliff Click Gil Tene Michael Wolf

Azul Systems, Inc.
1600 Plymouth Street
Mountain View, CA 94043
{cliffc,gil,wolf}@azulsystems.com

ABSTRACT

Modern transactional response-time sensitive applications have run into practical limits on the size of garbage collected heaps. The heap can only grow until GC pauses exceed the response-time limits. Sustainable, scalable concurrent collection has become a feature worth paying for.

Azul Systems has built a custom system (CPU, chip, board, and OS) specifically to run garbage collected virtual machines. The custom CPU includes a *read barrier* instruction. The read barrier enables a highly concurrent (no stop-the-world phases), parallel and compacting GC algorithm. The Pauseless algorithm is designed for uninterrupted application execution and consistent mutator throughput in every GC phase.

Beyond the basic requirement of collecting faster than the allocation rate, the Pauseless collector is never in a "rush" to complete any GC phase. No phase places an undue burden on the mutators nor do phases race to complete before the mutators produce more work. Portions of the Pauseless algorithm also feature a "self-healing" behavior which limits mutator overhead and reduces mutator sensitivity to the current GC state.

We present the Pauseless GC algorithm, the supporting hardware features that enable it, and data on the overhead, efficiency, and pause times when running a sustained workload.

Categories and Subject Descriptors

D.3.4 [Processors] – Memory management, D.3.3 [Language Constructs and Features] – Dynamic storage management,

General Terms

Languages, Performance, Design, Algorithms.

Keywords

Read barriers, memory management, garbage collection, concurrent GC, Java, custom hardware

1. INTRODUCTION

Many of today's enterprise applications are based on garbage collected virtual machine environments such as Java and .NET. Most have response time sensitive components – for example, a person may be waiting for a web page to load, or a credit-card swipe needs to complete. Stopping for an inopportune GC pause can lead to unacceptable response times. For these applications it is unacceptable for collectors to drive high average throughput numbers at the expense of occasional poor response times.

These enterprise applications need a low-pause time collector (pauses on the order of human reflexes, 10-100ms) that can handle very large Java programs (heap sizes from 100MB to 100GB) and highly concurrent workloads (100s of concurrent mutator threads). Such a collector needs to perform consistently and predictably over long periods of time, rather than simply excel at short time-bursts of workload.

Many modern garbage collectors rely on *write barriers* imposed on mutator heap writes, to keep track of references between different heap regions. This enables an efficient generational or region-based GC and is widely used in many garbage-collected languages including most production Java implementations. *Read barriers*, on the other hand, are rarely used in production systems despite a wealth of academic research because of the high mutator cost they usually incur.

Azul Systems has built a custom system (CPU, chip, board, and OS) specifically to run garbage collected virtual machines. The custom CPU includes a *read barrier* instruction. The read barrier enables a highly concurrent, parallel and compacting GC algorithm. The Pauseless GC algorithm is simple, efficient (low mutator overhead), and has no Stop-The-World pauses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'05, June 11–12, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-047-7/05/0006...$5.00.

2. RELATED WORK

The idea of garbage collection has been around for a long time [22][13][16][11]. We do not attempt to summarize all relevant GC work and instead we refer the reader to several GC surveys [30][31], and highlight a few papers.

GC pauses and their unpredictable impact on mutators was the driving force behind the early work on concurrent collectors [26][5]. The expectation of the time was that special GC hardware would shortly be feasible and commonplace. This early work required such extensive fine-grained synchronization that it would only be feasible on dedicated hardware. GC hardware continues to be proposed to this day [23][29][24][18][20].

The idea of using common page-protection hardware to support GC has also been around awhile [2]. Both Appel [2][3] and Ossia [25] protect pages that may contain objects with non-forwarded pointers (initially all pages). Accessing a protected page causes

an OS trap which the GC handles by forwarding all pointers on that page, then clearing the page protection. Appel does the forwarding in kernel-mode, Ossia maps the physical memory with a second unprotected virtual address for use by GC. Our Pauseless collector protects pages that contain objects being moved, instead of protecting pages that contain pointers to moved objects. This is a much smaller page set, and the pages can be incrementally protected. Our read-barrier allows us to intercept and correct individual stale references, and avoids blocking the mutator to fix up entire pages. We also support a special GC protection mode to allow fast, non-kernel-mode trap handlers that can access protected pages.

The idea of an incremental collector (via reference counting) is not new either [15]. Incremental collection seeks to reduce pause time by spreading the collection work out in time, finely interleaving GC work with mutator work. Because reference counting is expensive, and indeed all barriers (reference counting typically involves a write barrier) impose some mutator cost there is considerable research in reducing barrier costs [6][21][8][9]. Having the read-barrier implemented in hardware greatly reduces costs. In our case the typical cost is roughly that of a single cycle ALU instruction.

Incremental and low-pause-time collectors are becoming very popular again – partly because embedded devices have grown in compute power to the point where it's feasible to run a garbage collected language on them [19]. Metronome is an example of a modern low-pause time collector for a uniprocessor embedded system, and the pause times reported for Metronome are indeed smaller than those reported here [4]. However, Metronome as currently described is single-threaded and large business-class applications have enough mutators to overwhelm any single-threaded collector. Pauseless is fully parallel and can add GC worker threads at any time. Metronome requires an oracle to predict the future GC needs of running applications; this oracle is easily supplied in embedded systems with a fixed application set (the engineer runs the finite application set and measures GC consumption). Servers typically do not have a fixed application set and GC requirements are highly unpredictable. Metronome mutator utilization is around 50%. In contrast our mutator utilization is closer to 98%, we use extra CPUs to do the collection work. In exchange, Metronome provides hard real-time guarantees while we provide only soft real-time guarantees.

Our read-barrier is used for Baker-style relocation [5][23], where the loaded value is corrected before the mutator is allowed to use it. We focus collection efforts on regions which are known to be mostly dead, similar to Garbage-First [14]. Our mark phase uses an incremental update style instead of *Snapshot-At-The-Beginning* (SATB) style [30]. SATB requires a modestly expensive write-barrier which first does a read (and generally a series of dependent tests). The Pauseless collector does not require a write barrier.

Concurrent GCs are available in most modern production JVMs; BEA's JRockit [7], SUN's HotSpot [28] and IBM's production JVM [27] all have concurrent collectors and we tested with the latest available versions of Java 1.4 from each vendor. However, in all cases these collectors are not the defaults. They appear to not be as stable as the parallel collectors and they sometimes put high overheads on mutator threads. For some of these collectors, worse-case transaction times were no better than the default collectors.

3. HARDWARE SUPPORT

3.1 Background

Azul Systems has built a custom system (CPU, chip, board, and OS) specifically to run garbage collected virtual machines such as Java; the JVM is based on SUN's HotSpot [28]. We describe actual production hardware, which had real costs to design, develop and debug. Thus we were strongly motivated to design simple and cost-effective hardware. In the end, the custom GC hardware we built was quite minor.

The basic CPU core is a 64-bit RISC optimized to run modern managed languages like Java. It makes an excellent JIT target but does not directly execute Java bytecodes. Each chip contains 24 CPUs, and up to 16 such chips can be made cache-coherent; the maximum sized system has 384 CPU cores and 256G of memory in a flat, symmetric memory space. The box runs a custom OS and can run as many JVMs as memory allows. A single JVM can dynamically scale to use all CPUs and all available memory.

The hardware supports a number of fast user-mode trap handlers. These trap handlers can be entered and left in a handful of clock cycles (4-10, depending) and are frequently used by the GC algorithm; fast traps are key. The hardware also supports a fast cooperative preemption mechanism via interrupts that are taken only on user-selected instructions.

3.2 OS-level Support

The hardware TLB supports an additional privilege level, the GC-mode, between the usual user- and kernel-modes. Usage of this GC-mode is explained in the GC algorithm section. Several of the fast user-mode traps start the trap handler in GC-mode instead of user-mode. The TLB also supports 1 megabyte pages; the 1M page thus becomes the standard unit of work for the Pauseless GC algorithm and appears frequently below.

The TLB is managed by the OS in the usual ways, with normal kernel-level TLB trap handlers being invoked when normal loads and stores fail an address translation. Setting the GC privilege mode bit is done by the JVM via calls into the OS. TLB violations on GC-protected pages generate fast user-level traps instead of OS level exceptions.

HotSpot supports a notion of GC *safepoints*, code locations where we have precise knowledge about register and stack locations [1]. The hardware supports a fast cooperative preemption mechanism via interrupts that are taken only on user-selected instructions, allowing us to rapidly stop individual threads only at safepoints. Variants of some common instructions (e.g., backwards branches, function entries) are flagged as safepoints and will check for a pending per-CPU safepoint interrupt. If a safepoint interrupt is pending the CPU will take an exception and the OS will call into a user-mode *safepoint-trap* handler. The running thread, being at a known safepoint, will then save its

state in some convenient format and call the OS to yield. When the OS wants to preempt a normal Java thread, it sets this bit and briefly waits for the thread to yield. If the thread doesn't report back in a timely fashion it gets preempted as normal.

The result of this behavior is that nearly all stopped threads are at GC safepoints already. Achieving a global safepoint, a *Stop-The-World* (STW) pause, is much faster than *patch-and-roll-forward* schemes [1] and is without the runtime cost normally associated with software polling schemes. While the algorithm we present has no STW pauses, our current implementation does. Hence it's still useful to have a fast stopping mechanism.

We also make use of *Checkpoints*, points where we cannot proceed until all mutator threads have performed some action. In a Checkpoint each mutator reaches a GC safepoint, does a small amount of GC-related work and then carries on. Blocked threads are already at GC safepoints; GC threads perform the action on their behalf. In a STW pause, all mutators must reach a GC safepoint before any of them can proceed; the pause time is governed by the slowest thread. In a Checkpoint, running threads are never idled and the GC work is spread out in time. The same hardware and OS support is used for both STW pauses and Checkpoints.

3.3 Hardware Read Barrier

In addition to the standard RISC load/store instruction set, the CPUs have a few custom instructions to aid in object allocation and collection. In this paper we focus on the hardware *read barrier*. It is instructive to note that this barrier strongly resemble those from 20 years ago [23].

The read barrier performs a number of checks and is used in different ways during different GC phases. Its behavior is described briefly here, and then again in greater depth in the context of the GC algorithm in the next section. The read barrier is issued after a load instruction and executes in 1 clock. There is a standard load-use penalty which the compiler attempts to schedule around.

The read barrier "looks like" a standard load instruction, in that it has a base register, an offset and a value register. The base and offset are not used by the barrier checks but are presented to the trap handler and are used in "self healing". The value in the value register is assumed to be a freshly loaded *ref*, a heap pointer, and is cycled through the TLB just like a base address would be. If the ref refers to a GC-protected page a fast user-mode trap handler is invoked, hereafter called the *GC-trap*. The read barrier ignores null refs. Unlike a Brooks-style [10] indirection barrier there is no null check, no memory access, no load-use penalty, no extra word in the object header and no cache footprint. This behavior is used during the concurrent Relocate phase.

We also steal 1 address bit from the 64-bit address space; the hardware ignores this bit (masks it off). This bit is called the *Not-Marked-Through* (NMT) bit and is used during the concurrent Marking phase. The hardware maintains a desired value for this bit and will trap to the *NMT-trap* if the ref has the wrong flavor. Null refs are ignored here as well.

Note that the read barrier behavior can be emulated on standard hardware at some cost. The GC protection check can be emulated with standard page protection and the read barrier emulated with a dead load instruction. The NMT check can be emulated by double-mapping memory and changing page protections to reflect the expected NMT bit value. However, using the TLB to check ref privileges means that a failure will trigger a kernel-level TLB trap instead of a fast user-mode trap. Turning this into a user-mode trap will generally have some substantial cost and may require altering the OS. Our read barrier instruction will not trap on a null ref, and null refs are quite common. Emulating this on standard hardware will require a conditional test in the barrier code or mapping page 0. This in turn precludes using normal memory operations from doubling as null-pointer checks, a common optimization in modern JVMs.

4. THE PAUSELESS GC ALGORITHM

The Pauseless GC Algorithm is divided into three main phases: Mark, Relocate and Remap. Each phase is fully parallel and concurrent. Mark bits go stale; objects die over time and the mark bits do not reflect the changes. The Mark phase is responsible for periodically refreshing the mark bits. The Relocate phase uses the most recently available mark bits to find pages with little live data, to relocate and compact those pages and to free the backing physical memory. The Remap phase updates every relocated pointer in the heap.

There is no "rush" to finish any given phase. No phase places a substantial burden on the mutators that needs to be relieved by ending the phase quickly. There is no "race" to finish some phase before collection can begin again – Relocation runs continuously and can immediately free memory at any point. Since all phases are parallel, GC can keep up with any number of mutator threads simply by adding more GC threads. Unlike other incremental update algorithms, there is no re-Mark or final-Mark phase; the concurrent Mark phase will complete in a single pass despite the mutators busily modifying the heap. GC threads do compete with mutator threads for CPU time. On Azul's hardware there are generally spare CPUs available to do GC work. However, "at the limit" some fraction of CPUs will be doing GC and will not be available to the mutators.

Each of the phases involves a "self-healing" aspect, where the mutators immediately correct the cause of each read barrier trap by updating the ref in memory. This assures the same ref will not trigger another trap. The work involved varies by trap type and is detailed below. Once the mutators' working sets have been handled they can execute at full speed with no more traps. During certain phase shifts mutators suffer through a "trap storm", a high volume of traps that amount to a pause smeared out in time. We measured the trap storms using Minimum Mutator Utilization, and they cost around 20ms spread out over a few hundred milliseconds.

The algorithm we present has no *Stop-The-World* (STW) pauses, no places where all threads must be simultaneously stopped. However, for ease of engineering into the existing HotSpot JVM our implementation includes some STWs. We feel these STWs can be readily engineered to have pause times below standard OS context-switch times, where a GC pause will be indistinguishable

from being context switched by the OS. We will mention where the implementation differs from theory as the phases are described.

4.1 Mark Phase

The Mark phase is a parallel and concurrent incremental update (not SATB) marking algorithm [17], augmented with the read barrier. The Mark phase is responsible for *marking* all live objects, tagging live objects in some fashion to distinguish them from dead objects. In addition, each ref has it's NMT bit set to the expected value. The Mark phase also gathers per-1M-page liveness totals. These totals give a conservative estimate of live data on a page (hence a guaranteed amount of reclaimable space) and are used in the Relocate phase.

The basic idea is straightforward: the Marker starts from some root set (generally static global variables and mutator stack contents) and begins marking reachable objects. After marking an object (and setting the NMT bit), the Marker then marks-through the object – recursively marking all refs it finds inside the marked object. Extensions to make this algorithm parallel have been previously published [17]. Making marking fully concurrent is a little harder and the issues are described further below.

4.2 Relocate Phase

The Relocate phase is where objects are relocated and pages are reclaimed. A page with mostly dead objects is made wholly unused by relocating the remaining live objects to other pages. The Relocate phase starts by selecting a set of pages that are above a given threshold of sparseness. Each page in this set is protected from mutator access, and then live objects are copied out. Forwarding information tracking the location of relocated objects is maintained outside the page.

If a mutator loads a reference to a protected page, the read-barrier instruction will trigger a GC-trap. The mutator is never allowed to use the protected-page reference in a language-visible way. The GC-trap handler is responsible for changing the stale protected-page reference to the correctly forwarded reference.

After the page contents have been relocated, the Relocate phase frees the **physical** memory; it's contents are never needed again. The physical memory is recycled by the OS and can immediately be used for new allocations. **Virtual** memory cannot be freed until no more stale references to that page remain in the heap, and that is the job of the Remap phase.

As hinted at in Figure 1, a Relocate phase runs constantly freeing memory to keep pace with the mutators' allocations. Sometimes it runs alone and sometimes concurrent with the next Mark phase.

4.3 Remap Phase

During the Remap phase, GC threads traverse the object graph executing a read barrier against every ref in the heap. If the ref refers to a protected page it is stale and needs to be forwarded, just as if a mutator trapped on the ref. Once the Remap phase completes no live heap ref can refer to pages protected by the previous Relocate phase. At this point the virtual memory for those pages is freed.

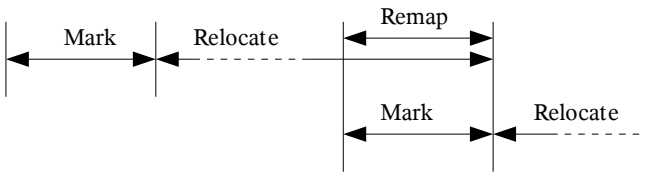

Figure 1: The Complete GC Cycle

Since both the Remap and Mark phases need to touch all live objects, we fold them together. The Remap phase for the current GC cycle is run concurrently with the Mark phase for the next GC cycle, as shown in Figure 1.

The Remap phase is also running concurrently with the 2nd half of the Relocate phase. The Relocate phase is creating new stale pointers that can only be fixed by a complete run of the Remap phase, so stale pointers created during the 2nd half of *this* Relocate phase are only cleaned out at the end of the *next* Remap phase. The next few sections will discuss each phase in more depth.

5. MARK PHASE

The Mark phase begins by initializing any internal data structures (e.g., marking worklists) and clearing this phase's *mark-bits*. Each object has two mark-bits, one indicating whether the ref is reachable (hence live) in this GC cycle, and one for it's state in the prior cycle.[1]

The Mark phase then marks all global refs, scans each threads' *root-set*, and flips the per-thread expected NMT value. The root-set generally includes all refs in CPU registers and on the threads' stacks. Running threads cooperate by marking their own root-set. Blocked (or stalled) threads get marked in parallel by Mark-phase threads. This is a Checkpoint; each thread can immediately proceed after it's root set has been marked (and expected-NMT flipped) but the Mark phase cannot proceed until all threads have crossed the Checkpoint.

After the root-sets are all marked we proceed with a parallel and concurrent marking phase [17]. Live refs are pulled from the worklists, their target objects marked live and their internal refs are recursively worked on. Note that the markers ignore the NMT bit, it is only used by the mutators. When an object is marked live, its size is added to the amount of live data in it's 1M page (only large objects are allowed to span a page boundary and they are handled separately, so the live data calculation is exact). This phase continues until the worklists run dry and all live objects have been marked.

New objects created by concurrent mutators are allocated in pages which will not be relocated in this GC cycle, hence the state of their live bits is not consulted by the Relocate phase. All refs being stored into new objects (or any object for that matter) have either already been marked or are queued in the Mark phase's worklists. Hence the initial state of the live bit for new objects doesn't matter for the Mark phase.

5.1 The NMT Bit

One of the difficulties in making an incremental update marker is that mutators can "hide" live objects from the marking

[1] We use bitmaps for the marks, they're cheap to clear and scan.

threads. A mutator can read an unmarked ref into a register, then clear it from memory. The object remains live (because its ref is in a register) but not visible to the marking threads (because they are past the mutator stack-scan step). The unmarked ref can also be stored down into an already marked region of the heap. This problem is typically solved by requiring another STW pause at the end of marking. During this second STW the marking threads revisit the root-set and modified portions of the heap and must mark any new refs discovered. Some GC algorithms have used a SATB invariant to avoid the extra STW pause. The cost of SATB is a somewhat more expensive write-barrier; the barrier needs to read and test the overwritten value.

Instead of a STW pause or write-barrier we use a read barrier and require the mutators do a little GC work when they load a potentially unmarked ref by taking an NMT-trap. We get the trapping behavior by relying on the read-barrier and the *Not-Marked-Through* bit: a bit we steal from each ref. Refs are 64-bit entities in our system representing a vast address space. The hardware implements a smaller virtual address space; the unused bits are ignored for addressing purposes. The read-barrier logic maintains the notion of a desired value for the NMT bit and will trap if it is set wrong. Correctly set NMT bits cost no more than the read-barrier cost itself. The invariant is that refs with a correct NMT have definitely been communicated to the Marking threads (even if they haven't yet been marked through). Refs with incorrect NMT bits *may* have been marked through, but the mutator has no way to tell. It informs the marking threads in any case.

If a mutator thread loads and read-barriers a ref with the NMT bit set wrong, it has found a potentially unvisited ref. The mutator jumps to the NMT-trap handler. In the NMT-trap handler the loaded value has it's NMT bit set correctly. The ref is recorded with the Mark phase logic.[2] Then the corrected ref is stored back into memory. Since the ref is changed in memory, that particular ref will not cause a trap in the future.

This "self-healing" idea is key: without it a phase-change would cause all the mutators to take continuous NMT traps until the Marker threads can get around to flipping the NMT bits in the mutators' working sets. Instead, each mutator flips its own working set as it runs. After a short period of high-intensity trapping (a "trap storm") the working set is converted and the mutator proceeds at its normal pace. During the steady-state portion of the Mark phase, mutators take only rare traps as their working set slowly migrates.

Changing the ref in memory amounts to a store, even if the stored value is Java-language-equivalent to the original value. The store is transparent to the Java semantics of the running thread, but the store is visible to other threads: without some care it might stomp over another thread's store effectively reversing it. Instead of unconditionally storing, the trap handler uses a *compare-and-swap* (CAS) instruction to only update the memory if it hasn't changed since the trap. If the CAS fails the handler returns the value currently in memory (**not** the value originally loaded) and the read barrier is repeated.

[2] Actually, they are batched for efficiency.

5.2 The NMT Bit and The Initial Stack-Scan

Refs in mutators' root-set have already passed any chance for running a read-barrier. Hence the initial root-set stack-scan also flips the NMT bits in the root-set. Since the flipping is done with a Checkpoint instead of a STW pause, for a brief time different threads will have different settings for the NMT desired value. It is possible for two threads to *throb*, to constantly compete over a single ref's desired value NMT value via trapping and updating in memory. This situation can only last a short period of time, until the unflipped thread passes the next GC safepoint where it will trap, flip its stack, and cross the Checkpoint.

Note that it is **not** possible for a single thread to hold the same ref twice in its root-set with different NMT settings. Hence we do not suffer from the pointer-equality problem; if two refs compare as bitwise not-equal, then they are truly unequal.

5.3 Finishing Marking

When the marking threads run out of work, Marking is nearly done. The marking threads need to close the narrow race where a mutator may have loaded an unmarked ref (hence has the wrong NMT bit) but not yet executed the read-barrier. Read-barriers never span a GC safepoint, so it suffices to require the mutators cross a GC safepoint without trapping. The Marking pass requests a Checkpoint, but requires no other mutator work. Any refs discovered before the Checkpoint ends will be concurrently marked as normal. When all mutators complete the Checkpoint with none of them reporting any new refs, the Mark phase is complete. If new refs are reported the Marker threads will exhaust them and the Checkpoint will repeat. Since no refs can be created with the "wrong" NMT-bit value the process will eventually complete.

6. RELOCATE AND REMAP PHASES

The Relocate phase is where objects get relocated and compacted, and unused pages get freed. Recall that the Mark phase computed the amount of live data per 1M page. A page with zero live data can obviously be reclaimed. A page with only a little live data can be made wholly unused by relocating the live objects out to other pages.

As hinted at in Figure 1, a Relocate phase is constantly running, continuously freeing memory at a pace to stay ahead of the mutators. Relocation uses the current GC-cycle's mark bits. A cycle's Relocate phase will overlap with the next cycle's mark phase. When the next cycle's Mark phase starts it uses a new set of marking bits, leaving the current cycle's mark bits untouched.

The Relocate phase starts by finding unused or mostly unused pages. In theory full or mostly full pages can be relocated as well but there's little to be gained. Figure 2 shows a series of 1M heap pages; live object space is shown textured. There is a ref coming from a fully live page into a nearly empty page. We want to relocate the few remaining objects in the "Mostly Dead" page and compact them into a "New, Free" page, then reclaim the "Mostly Dead" page.

Next the Relocate phase builds side arrays to hold forwarding pointers. The forwarding pointers cannot be kept in the old copy of the objects because we will reclaim the physical storage immediately after copying and long before all refs are remapped.

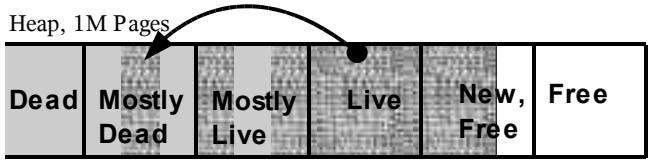

Figure 2: Finding sparsely populated pages

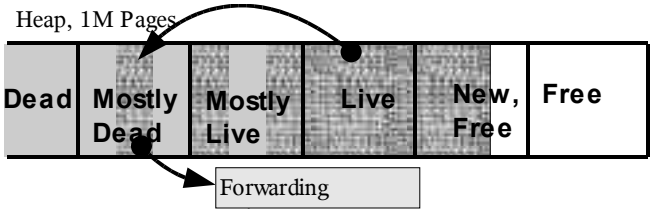

Figure 3: Side Arrays and TLB Protection

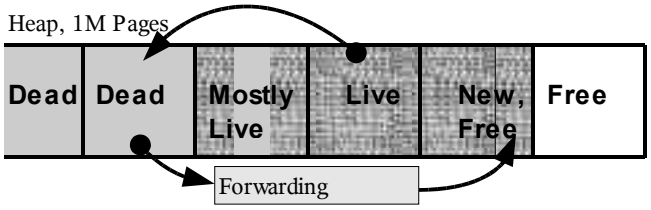

Figure 4: Copying live data out

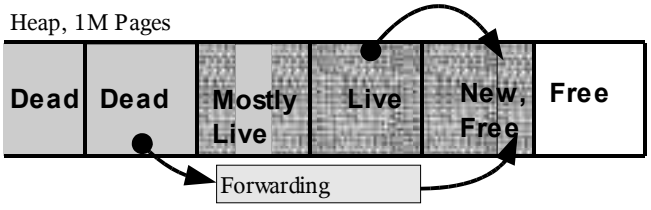

Figure 5: Updating stale refs

The side array data isn't large because we relocate sparse pages. We implement it as a straightforward hash table. Figure 3 shows the side array.

The Relocate phase then GC-protects the "Mostly Dead" page, shown in gray, from the mutators. Objects in this page are now considered stale; no more modifications of these objects are allowed. If a mutator loads a ref into the protected page, it's read-barrier will now take a GC-trap.

Next the live objects are copied out and the forwarding table is modified to reflect the objects' new locations as shown in Figure 4. Copying is done concurrently with the mutators; the read-barrier keeps the mutators from seeing a stale object before it has finished moving. Live objects are found using the most recent mark-bits available and sweeping the page.

Once copying has completed, the **physical** memory behind the page is freed. Virtual memory cannot be reclaimed until there are no more stale refs pointing into the freed page. Stale refs are left in the heap to be lazily discovered by running mutators using the read-barrier, and will be completely updated in the next Remap phase. Freed physical memory is immediately recycled by the OS and may be handed out to this or another process. After freeing memory, the GC threads are idled until the next need to relocate and free memory, or until the next Mark and Remap phase begins.

6.1 Read-Barrier Trap Handling

If a mutator's read-barrier GC-traps, then the mutator has loaded a stale ref. The GC-trap handler looks up the forwarding pointer from the side arrays and places the correct value both in the register and in memory, as shown in Figure 5. Similarly to the NMT trap handler's "self-healing" behavior, updating the ref in memory is crucial to performance: it keeps the same stale ref from trapping again. As before, the memory update is done with a CAS to avoid stomping a racing store from another thread.

It is also possible that the needed object has not yet been copied. In this case the mutator will do the copy on behalf of the GC thread – since the mutator is otherwise blocked from forward progress. The mutator can read the GC-protected page because the trap handler runs in the elevated GC-protection mode. If the mutator must copy a large object, it may be stalled for a long time. This normally isn't an issue: pages with a lot of live data are not relocated and a ½-page sized object (512K) can be copied in about 1ms.

6.2 Other Relocate Phase Actions

At the time we protected pages, running mutators might have stale refs in their root-set. These are already past their read-barrier and thus won't get directly caught. The mutators scrub any existing stale refs from their root-set with a Checkpoint. Relocation can start when the Checkpoint completes.

The cost to modify the TLB protections (a kernel call and a system-wide TLB shoot-down) and scrubbing the mutators' stacks is the same for one page as it is for many. We batch up these operations to lower costs, and typically protect (and relocate and free) a few gigabytes at a time.

Notice that there is no "rush" to finish the Relocation phase; we need only relocate and free pages at a pace to keep ahead of the mutators. Also notice it is unlikely that a mutator stalls on an unmoved stale object. Relocated pages contain only a few older objects, most likely they have moved out of the mutator's working set. Virtual memory is not freed immediately, but we have lots of that. The final step of scrubbing all stale refs and reclaiming virtual memory is the job of the Remap phase.

6.3 The Remap Phase

The Remap phase updates all stale refs with their proper forwarded pointers. It must visit every ref in the heap to find all the stale ones. As mentioned before it runs in lockstep with the next GC cycle's Mark phase; the one piece of visitor logic does both the stale ref check and NMT check.

At the end of the Remap phase, all pages that were protected before the start of the Remap phase have now been completely scrubbed. No more stale refs to those pages remain so those virtual memory pages can now be reclaimed. We also free the side arrays at this time, and a GC cycle is complete.

7. REALITY CHECK

Our implementation is a rapidly moving work-in-progress. As of this writing it suffers from a few STW pauses not required by the Pauseless GC algorithm. Over time we hope to remove these

STWs or engineer their maximum time below an OS time-slice quanta. We have proposed solutions for each one, and report pauses experienced by the current implementation on the 8-warehouse 20-minute pseudo-JBB run described in Section 8.

7.1 At the Mark Phase Start

At the start of the Mark phase we stop all threads to flip the desired NMT state. We could flip the NMT bits via a Checkpoint; the cost would be some amount of NMT-bit *throbbing* (repeated NMT traps) on shared objects until all threads flip. Also, global shared resources (e.g., the SystemDictionary, JNI handles, locked objects) are marked in this STW. Engineering these to use a Checkpoint is straightforward.

The worse pause reported was 21ms and the average was 16ms.

7.2 At the Mark Phase End

At the end of the Mark phase we stop all threads and do (in parallel but not concurrent) soft ref processing, weak ref processing, and finalization. Java's soft and weak refs present a race between the collector nullifying a ref and the mutator "strengthening" the ref. We could process the refs concurrently by having the collector CAS down a null only when the ref remains not-marked-through. The NMT-trap handler already has the proper CAS'ing behavior – both the collector and the mutator race to CAS down a new value. If the mutator wins the ref is strengthened (and the collector knows it), and if the collector wins the ref is nullified (and the mutator only sees the null).

There are a number of other items handled at this STW that could be engineered to be concurrent, including class unloading and code-cache unloading. Again engineering these will be straightforward but tedious.

The worse pause reported was 16ms and the average was 7ms.

7.3 At the Relocation Phase Start

The mutators' root-sets need scrubbing when GC-protecting a page. There are two problems here: the TLB shoot-down isn't atomic and there are stale refs in the root-set. Since the TLB shoot-down is not atomic, for a brief period some mutators can be protected and not others. Unprotected mutators would continue to read and write the object directly, so protected mutators need to as well. However, reading and writing the protected object forces a GC-protection trap. Our current implementation stops all threads and performs a bulk TLB shoot-down and mutator root-set scrubbing under STW. This can be engineered to be concurrent and incremental in a straightforward manner.

We could use a Checkpoint to update the TLBs and scrub the root-sets. To maintain concurrency until all threads have passed the relocation Checkpoint, the read barrier's TLB trap handler is modified to wait for the Checkpoint to complete before proceeding with relocation or remapping and propagating a corrected ref in the mutator. Mutator threads that actually access refs in protected pages will then "bunch up" at the Checkpoint with other threads continuing concurrent execution past the Checkpoint. This effect is mitigated by the fact that we preferentially relocate sparse pages.

The worse pause reported was 19ms and the average was 5ms.

7.4 Relocate doesn't run during Mark/Remap

Right now we have not implemented a second set of mark bits to allow the Relocate phase to run concurrently with the next Mark/Remap phase [14]. This means we cannot free memory during the Mark/Remap phase. We have heuristics which predict how many pages the mutator will need during marking and we free that many (plus some pad) before marking begins. If we predict low, as can happen if the mutators suddenly "accelerate", the mutators will block until marking is complete. Engineering the overlapped Relocate/Mark phases will be straightforward. Additionally, we currently do not add threads dynamically in response to mutator acceleration. Each phase completes with a number of threads decided on at the phase start.

8. EXPERIMENTS

8.1 Methodology

The Pauseless algorithm is intended to lower pause times in large transaction-oriented programs running business logic. There are a limited number of representative Java benchmarks for this class of program. The most realistic and widely accepted is SpecJApp-Server '02 and '04. This benchmark is extremely difficult to setup, tune, or get reliable numbers out of. It is also very hard to normalize across different hardware. The much more simplistic SpecJBB benchmark has very well-structured (and unrealistic!) object lifetimes and is ideally suited for a generational collector.

In an effort to have both a reliable, understandable benchmark and one that is more representative of transactional programs, we added a large object cache to the standard SpecJBB benchmark. This cache represents, e.g., a Java bean cache, or an HTML request cache. For each transaction, 400 bytes were added to the cache and the oldest cached object was freed. This level of extra objects is enough to easily defeat targeted tuning of generational collectors to JBB.

We also removed the forced System.gc() between runs and increased the JBB run times from 2 minutes to 20 minutes.[3] In the standard benchmark it's common to never need a full collection during the timed portion of the run. In practice, these large business applications must run in a steady-state mode without an untimed window every 2 minutes for a System.gc().

All runs were done with 8 warehouses, i.e. 8 concurrent threads doing benchmark work. We added "-Xmx1536m", allowing a maximum heap size of 1.5G, which is about twice the average size of the live data. We added "-server" to the SUN JVMs. For the concurrent GC timing runs, we added whatever flag was appropriate to trigger using the concurrent collector for that JVM. For the IBM JVM, it was "-Xgcpolicy:optavgpause". For the BEA JVM, it was "-Xgcprio:pausetime". For the SUN JVM, it was "-XX:+UseConcMarkSweepGC -XX:+UseParNewGC". For the Azul JVM, concurrent collection is the default and no flags are needed. For the non-concurrent GC timing runs we used the best parallel (throughput-oriented) collector available. This is the default for the IBM and BEA JVMs, for the SUN JVM we added "-XX:+UseParallelGC". We used no other flags.

[3] Except for IBM's concurrent collector which was unable to run the full 20 minutes; we used a 10 minute run for it.

We ran the IBM and SUN JVMs on a 2-way 3.2Ghz hyper-threaded Xeon with 2G of physical memory, running a Red Hat Linux 2.6 kernel. Unfortunately, the BEA JVM didn't run on this version of Linux so it was run on a 1-way 2.4Ghz hyper-threaded P4 with 512M of physical memory running Windows 2000. The BEA JVM heap was limited to 425M to avoid paging. The simulated object cache added about 40M of long-lived live data per warehouse; 425M isn't a large enough heap to run with 8 warehouses. We limited the BEA JVM to 3 warehouses, keeping the proportion of heap devoted to long-lived data about the same. We also ran the SUN JVM in 64-bit mode on a 2-way 1.2Ghz US3 with 4G of physical memory running Solaris 9. We attempted to run on an older 24-CPU Sparc (450Mhz US2). Here we hoped the Sparc would use the spare CPUs to good effect. However, the single-threaded concurrent collector could not keep up with the mutators and the benchmark suffered numerous 12-second full-GC pauses. On the 2-CPU Sparc, a single concurrent collector thread could use up to half the total CPU resources in order to keep up. We report the superior 2-CPU Sparc scores, although we would like to have reported scores from another high-CPU count machine. The Azul JVM is a 64-bit JVM running on a 16-chip (384-CPU) Azul appliance with 128G of physical memory. As before, we limited heap size to 1.5G. Only 8 CPUs are used to run the actual benchmark, with a handful more running the Pauseless collection and doing background JIT compiles.

We decided to NOT report SpecJBB score, which is reported in units of transactions/second, both because our run is not Spec-compliant and because of the wide variation in hardware and JIT quality. Even on the same hardware, the JITs from different vendors produce code of substantially different quality. For the same 20 minute run, we saw JVMs execute between 15 million and 30 million transactions. While transaction throughput is an important metric, this paper is focused on removing the biggest reason for transaction time variability. We report transaction times instead.

8.2 Transaction Times

We measured both transaction times and GC pause times reported with "-verbose:gc". We feel that transaction times represent a more realistic measure than direct GC pauses as they more closely correspond to "user wait time".

Transaction times were gathered into buckets by duration, building a histogram. Duration was measured with Java's currentTimeMillis() and so is limited to millisecond resolution. Most transactions take 0 or 1 milliseconds, so we did not gather accurate times for these fast transactions. However, we are more interested in the slow transactions. All the collectors except Pauseless had a significant fraction of transactions take 100-300ms (100 times slower than the fast transactions), with spikes to 1-4 seconds. We kept per-millisecond buckets from 0ms to 31ms. After that we grew the buckets by powers-of-2 with halves: 32-47ms, 48-63ms, 64-95ms, 96-127ms, and so on up to 16sec. This allowed us to compute the bucket index with a few shifts. Buckets were replicated per thread to avoid coherency costs then totaled together at the end of the run.

A transaction that reports as taking 0ms clearly takes some finite time. The 0ms bucket's average transaction time is assumed to be 0.33ms, and the 1ms bucket's average transaction time is assumed to be 1.33ms. This is the largest source of measurement error we have. Almost no transactions landed in the 3ms to 30ms buckets, so a measurement error of up to 1ms in those buckets will not alter the data in any substantial way.

For all other buckets we simply totaled time for that bucket. We summed the total transaction times (time per bucket by transactions in the bucket), and report the percentage of total transaction time spent on transactions of each duration.

Figure 6 shows how many transactions the various JVMs kept in the 0ms and 1ms range (0ms is the low bar, 1ms is the middle bar). The Pauseless algorithm keeps 87% (99.5%) of total transaction time spent in transactions of 1ms (2ms) or less; the other JVMs vary between 80% down to 50%. The concurrent version from each vendor faired slightly worse than the parallel collectors, showing a slightly higher percentage of total time spent in slow transactions.

Figure 7 shows cumulative transaction times (not wall-clock time, which was 20 minutes) vs. transaction duration. Times are cumulative, reaching 1.00 (100% of total transaction time) at the top edge. Transaction duration runs across the bottom in a log scale. Lines that approach 1.00 quicker are better, representing a greater percentage of processing time spent in fast transactions.

We can see a couple of trends in this chart. Pauseless again does quite well, with essential 100% of time spent in fast transactions and a worst-case transaction time of 26 milliseconds. The other JVMs are roughly grouped into pairs with the parallel throughput collector line being slightly higher than the concurrent collector line for most of the chart. The lines cross as we near 100% of time and the slowest transactions; the concurrent collectors generally have smaller worst-cast times than the throughput collectors.

Table 1 shows the worse-case transaction times. The Pauseless algorithm's worse-case transaction time of 26ms is over 45 times better than the next JVM, BEA's parallel collector. Average transaction times are remarkable similar given the wide variation in hardware used.

Table 1: Worst-case and average times, in ms

	Azul txu con	*IBM x86 con*	*SUN x86 con*	*SUN sun con*	*BEA x86 con*	*IBM x86 par*	*SUN x86 par*	*SUN sun par*	*BEA x86 par*
Trans	26	1245	1277	1674	1281	1419	3195	5376	1172
Pause	21	526	210	544	230	734	2217	3953	562
Ratio	1.24	2.37	6.08	3.08	5.57	1.93	1.43	1.36	2.09
Avg Trans	0.65	0.60	0.71	0.93	0.53	0.57	0.71	0.82	0.52
Pause	9.4	137	63	71	70	414	317	704	348

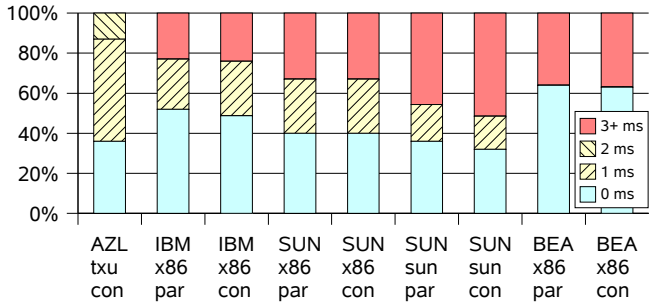

Figure 6: Short transaction times (0,1,2 ms) as a % of total

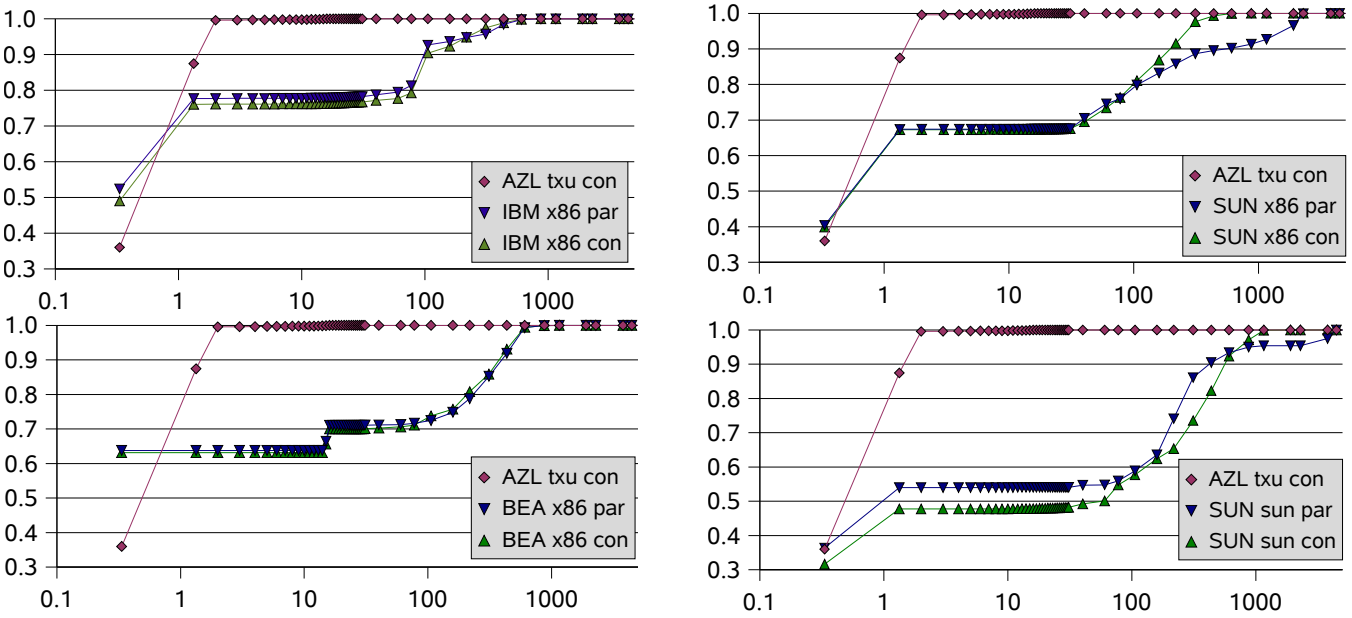

Figure 7: Cumulative transaction times vs. duration (ms)

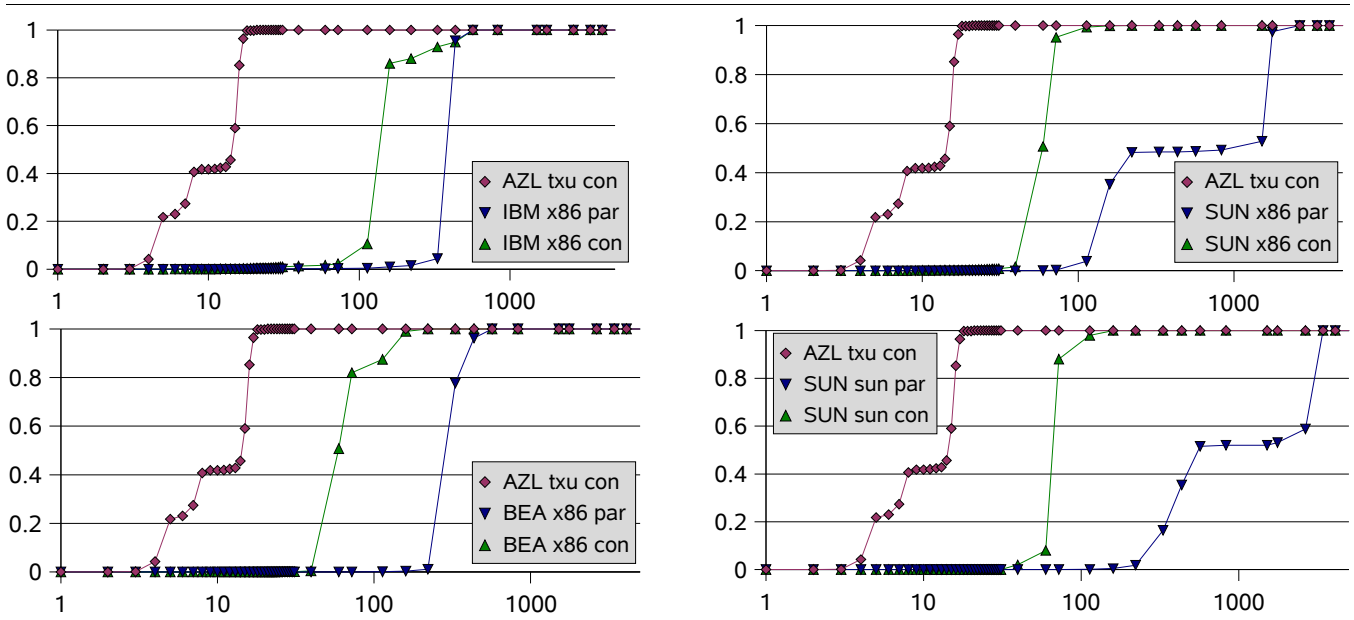

Figure 8: Reported pause times vs. duration (ms)

8.3Reported Pause Times

We collected GC pause times reported with "-verbose:gc". We summed all reported times and present a histogram of cumulative pause times vs. pause duration. Figure 8 shows the reported pauses. Most of the concurrent collectors consistently report pause times in the 40-50ms range; IBM's concurrent collector has 150ms as it's common (mode) pause time. As expected, the parallel collectors all do worse with the bulk of time spent in pauses ranging from 150ms to several seconds.

Table 1 also shows the ratio of worst-case transaction time and worst-case reported pause times. Note that JBB transactions are highly regular, doing a fixed amount of work per transaction. Changes in transaction time can be directly attributed to GC.[4] Several of the worse-case transactions are a full second longer than the worse-case pauses. We have some guesses as to why this is so:

It is possible that the concurrent collectors did not keep up with the allocation rate, stalling mutators until they caught up. Unfortunately, this information was not obvious from the "-verbose:gc" output. Also, during some phases of some concurrent GCs, the mutators pay a heavy cost while making forward progress. This amounts to an unreported pause smeared out in time. Sometimes the GC pauses come in rapid succession so that the same transaction will get paused several times. Perhaps the underlying OS timesliced the 8 mutator threads very poorly across the 4 hyper-threaded CPUs.

In any case, **reported pause times can be highly misleading**. The concurrent collectors other than Pauseless under-report their effects by 2x to 6x! The parallel collectors also under-report, but only by 30% to 100%. Based on this data, we encourage the GC research community to test the end-to-end effects of GC algorithms carefully.

We also attempted to gather Minimum Mutator Utilization figures [12], especially to track the "trap storm" effects. MMU reports the smallest amount of time available to the mutators in a continuous rolling interval. Since our largest pause was over 20ms there exists a 20ms interval where the mutators make no progress, so MMU@20ms is 0. Preliminary figures are in Table 2, and represent MMU figures for the entire 20 minute run worst case across all threads. Looking at the MMU@50ms figure, we see about 40ms of pause out of 50ms. We know that about 20ms of that is reported as an STW pause, so we assume the remaining 20ms is due to the trap storm.

Table 2: Minimum Mutator Utilization

MMU @20ms	*MMU@ 50ms*	*MMU@ 100ms*	*MMU@ 200ms*	*MMU@ 500ms*	*MMU@ 1000ms*	*MMU@ 2000ms*
0%	21%	40%	52%	67%	77%	84%

[4] We tested; all transactions are fast until the heap runs out. For the 64-bit JVMs we were able to test with a 64G heap.

9.Conclusions

Azul Systems has taken the rare opportunity to produce custom hardware for running a garbage collected language in a shipping product. This custom hardware enables a very potent garbage collection algorithm. Even though the individual Azul CPUs are slower than the high-clocking X86 P4's compared against, worse-case transaction times are over 45 times better and average transaction times are comparable.

Azul's Pauseless GC algorithm is a fully parallel and concurrent algorithm engineered for large multi-processor systems. It does not need any *Stop-The-World* pauses, no places where all mutator threads must be simultaneously stopped. Dead object space can be reclaimed at any point during a GC cycle; there are no phases where the GC algorithm has to "race" to finish some phase before the mutators run out of free space. Also there are no phases where the mutators pay a continuous high cost while running. There are brief "trap storms" at some phase shifts, but due to the "self-healing" property of the algorithm these storms appear to be low cost.

Azul's custom hardware includes a *read-barrier*, an instruction executed against every ref loaded from the heap. The read-barrier allows global GC invariants to be cheaply maintained. It checks for loading of potentially unmarked objects, preventing the spread of unmarked objects into previously marked regions of the heap. This allows the concurrent incremental update Mark phase to terminate cleanly without needing a final STW pause. The read-barrier also checks for loading stale refs to relocated objects and it does it cheaper than a Brooks' style indirection barrier.

Section 7, Reality Check, includes ongoing and future work. Another obvious and desirable feature is a generational variation of Pauseless. As presented, Pauseless is a single-generation algorithm. The entire heap is scanned in each Mark/Remap cycle. Because the algorithm is parallel and concurrent, and we have plentiful CPUs the cost is fairly well hidden. On a fully loaded system the GC threads will steal cycles from mutator threads, so we'd like the GC to be as efficient as possible. A generational version will only need to scan the young generation most of the time. The necessary hardware barriers already exists.

On a final note, we were quite surprised at the difference between reported pause times and the "user experience" delays seen by the transactions. We strongly encourage GC researchers and the production JVM providers to pay close attention to full GC algorithm costs, not just those costs that can easily have a timer-start/timer-stop wrapped around them.

10. REFERENCES

[1] Agesen, O. GC Points in a Threaded Environment. SMLI TR-98-70. Sun Microsystems, Palo Alto, CA. December 1998.

[2] Appel, A., Ellis, J., Li, K., Real-time concurrent collection on stock multiprocessors. In *1988 Conference on Programming Language Design and Implementation (PLDI)*, June 1988.

[3] Appel, A., Li, K., Virtual Memory Primitives for User Programs. In *1991 Conference on Architectural Support for Programming Languages and Operating System*, April 1991.

[4] Bacon, D., Cheng, P., Rajan, V. The Metronome: A simpler approach to garbage collection in real-time systems. In *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems*, Catania, Sicily, Nov. 2003.

[5] Baker, H., List processing in real time on a serial computer, *Communications of the ACM, Vol. 21*, 4, (April 1978),.280-294

[6] Barth, J. Shifting garbage collection overhead to compile time. *Communications of the ACM, Vol. 20,* 7 (July 1977), 513-518

[7] BEA Systems. 2003. BEA JRockit: Java for the Enterprise. White paper. BEA Systems, San Jose, CA.

[8] Blackburn, S., McKinley, K., In or Out? Putting Write Barriers in Their Place. In *Proceedings of the 2002 International Symposium on Memory Management*, Berlin, Germany, 2002.

[9] Blackburn, S., Hosking, A., Barriers: Friend or Foe? In *Proceedings of the 2004 International Symposium on Memory Management*, Vancouver, Canada, 2004.

[10] Brooks, R. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *1984 ACM Symposium on Lisp and Functional Programming*. (Aug. 1984) 256-262

[11] Cheney, C. A Nonrecursive List Compacting Algorithm. *Communications of the ACM, Vol. 13,* 11 (Nov. 1970), 677-678

[12] Cheng, P., Blelloch, G., A parallel, real-Time garbage collection. In *Conference on Programming Languages Design and Implementation (PLDI '01)*. Snowbird, Utah, June 2001

[13] Collins, G., A method for overlapping and erasure of lists. *Communications of the ACM, Vol. 3,* 12 (Nov. 1960), 655-657

[14] Detlefs, D., Flood, C., Heller, S., Printezis, T. Garbage-first garbage collection. In *Proceedings of the 2004 International Symposium on Memory Management*, Vancouver, Canada, 2004

[15] Deutcsh, P., Bobrow, D. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM, Vol. 19,* 9 (Sept. 1976), 522-527

[16] Fenichel, R., Yochelson, J. A LISP Garbage-Collector for Virtual Memory Systems. *Communications of the ACM, Vol. 12,* 11 (Nov. 1969), 611-612.

[17] Flood, C., Detlefs, D., Shavit, N., Zhang, C. Parallel Garbage Collection for Shared Memory Multiprocessors. In *2001 USENIX Java Virtual Machine Research and Technology Symposium (JVM '01)*. Monterey, CA, April 2001

[18] Goa, H., Nilsen, K. The real-time behavior of dynamic memory management in C++. In *Proceedings of the Real-Time Technology and Applications Symposium*. Chicago, IL, 1995

[19] Gosling, J., Bollela, G. *The Real-Time Specification for Java*. Addison-Wesley, Boston MA, 2000.

[20] Heil, T., Smith, J. Concurrent garbage collection using hardware-assisted profiling. In *Proceedings of the 2nd International Symposium on Memory Management*, Minneapolis, MN, 2000

[21] Hosking, A., Moss, E., Stefanovic, D., A comparative performance evaluation of write barrier implementations. In *Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*. Vancouver, Canada, Oct. 1992

[22] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM, Vol. 3,* 4 (April 1960), 184-195

[23] Moon, D. Garbage Collection in a Large LISP System. *1984 ACM Symposium on LISP and Functional Programming*. (Aug. 1984) 235-246

[24] Nilsen, K., Schmidt, W. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems (LOPLAS), Vol. 1,* 4 (Dec. 1992)

[25] Ossia, Y., Ben-Yitzhak, O., Segal, M., Mostly concurrent compaction for mark-sweep GC. In *Proceedings of the 2004 International Symposium on Memory Management*, Vancouver, Canada, 2004.

[26] Steele, G. Multiprocessing compactifying garbage collection. *Communications of the ACM, Vol. 18,* 9 (Sept. 1975), 495-508

[27] Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsd, H., and Nakatani, T. Overview of the IBM Java Just-in-Time Compiler, IBM Systems Journal, 39(1), 2000.

[28] Sun Microsystems. 2001. The Java HotSpot virtual machine. White paper. Sun Microsystems, Santa Clara, CA.

[29] Williams, I., Wolczko, M. An Object-Based Memory Architecture. In *Implementing Persistent Object Bases: Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114-130. Morgan Kaufmann Publishers, Inc., 1991.

[30] Wilson, P. Uniprocessor Garbage Collection Techniques. In *1992 Proceedings of the International Workshop on Memory Management (IWMM 92)*. Saint-Malo (France), 1992

[31] Wilson, P., Johnstone, M., Neely, M., Boles, D., Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management (IWMM 95)*, 1995