Concurrent Garbage Collection Using Hardware-Assisted Profiling

Timothy H. Heil James E. Smith
Electrical and Computer Engineering
University of Wisconsin - Madison
1415 Engineering Drive
{heilt,jes}@ece.wisc.edu

ABSTRACT

In the presence of on-chip multithreading, a Virtual Machine (VM) implementation can readily take advantage of *service threads* for enhancing performance by performing tasks such as profile collection and analysis, dynamic optimization, and garbage collection concurrently with program execution. In this context, a hardware-assisted profiling mechanism is proposed. The *Relational Profiling Architecture* (RPA) is designed from the top down. RPA is based on a relational model similar to the relational database model. Instructions selected for profiling produce a record of information. A simple *query engine* examines these records for patterns, and performs simple actions on matching records.

The power and flexibility of RPA is demonstrated by developing a concurrent generational garbage collector for Java. Detailed execution driven simulations show that this collector has an average runtime overhead of approximately 0.6%. The short pauses in the application required for synchronization with the garbage collector are at most 54 microseconds, given a 1GHz clock frequency.

Keywords

Concurrent garbage collection, hardware-assisted profiling, memory management

1. INTRODUCTION

As technology progresses, efficient hardware usage and the demand for higher levels of performance are leading to on-chip multi-threading, either through simultaneous multithreading (SMT) [33][38][37] or on-chip multiprocessing [4][23]. At the same time, the emergence of binary translation and virtual machine (VM) technologies are pointing to a re-definition of the traditional hardware/software interface.

Our research is targeted at this future environment and is centered on the development and application of *co-designed virtual machines* [28]. Co-designed VMs combine hardware and software to implement a virtual instruction set architecture (V-ISA). A co-designed virtual machine can use available on-chip multi-threading to implement *service threads* that perform such tasks as dynamic compilation, profile collection and processing, and garbage collection concurrent with program execution Properly developed hardware-assisted profiling can provide an efficient communication mechanism that allow VM service threads to monitor program behavior without slowing down program threads.

We employ the above concepts to implement a concurrent garbage collection (GC) mechanism supported by *Strata*, our experimental co-designed VM. GC executes on separate on-chip threads and uses hardware-assisted profiling to replace traditional inline store barriers. Detailed simulations show that this collector has an average run-time overhead of 0.6%, and the worst GC pause time is less than 0.2ms.

1.1 Co-designed Virtual Machines

Virtual machines execute programs coded in the V-ISA on hardware directly supporting another ISA, the implementation ISA (I-ISA). *Co-designed* virtual machines use a mix of hardware and software, designed together, to form a high performance implementation of the V-ISA. The I-ISA is designed to enable a clean, high performance microarchitecture that gets maximum benefit from the current generation chip technology. Conversely, the software portion of the VM is designed to improve processor performance through dynamic recompilation, special I-ISA instructions and hint bits, and pipeline tuning features that are visible through the I-ISA. In this context, the I-ISA becomes very fluid because the V-ISA provides the necessary binary compatibility. The System/38 [5] was a pioneer in this field, and there has been a flurry of recent co-designed VMs. These include Transmeta's Crusoe processors [18], Sun Microsystem's MAJC processor [30], and the DAISY [16] and BOA [17] research projects at IBM.

1.2 Thread-Level Parallelism and Virtual Machines

Widespread use of thread level parallelism (TLP) is likely to provide the next large gains in general purpose performance. Compared with instruction-level parallelism (ILP), TLP is relatively easy to support in hardware. ILP-oriented superscalar processor designs now consume enormous resources in terms of transistors, area, and power to reap even incremental gains in instructions per cycle. If processors were designed to exploit TLP, instead of ILP, much greater peak throughput could be obtained with the same resources. Of course, this requires that there are multiple threads of execution to exploit. Most recent superscalar designs focus on ILP primarily because multiple threads have only rarely been available in a general-purpose environment.

To appear: 2000 International Symposium on Memory Management, October 2000.

High performance VMs have the potential to change this. Many tasks that a VM performs are conceptually parallel to program execution. These include garbage collection, program profile collection and analysis, and dynamic optimization. Furthermore, each of these tasks can be parallelized itself. Hence, VMs have the potential for providing many service threads of execution, which collectively lead to much higher performance for the main program thread(s).

1.3 Profiling and inter-processor communication

An important aspect of multi-threaded co-designed VMs is the interaction of the main program thread(s) and service threads. There must be mechanisms for monitoring the performance of program threads and low-overhead communication mechanisms so that service threads can respond efficiently and correctly to the application's requirements.

Consequently, we made profiling a high priority and took a top-down approach to hardware-assisted profiling, first determining the properties of program execution that might need to be observed. The goal was to develop a mechanism flexible enough to meet not only current needs, but unanticipated future needs as well. From this we developed a framework for instruction-level profiling, which was then engineered to an efficient implementation. The result was the *Relational Profiling Architecture* (RPA), described in detail in Section 3.

A central outcome of this research is the realization that there is synergy between hardware-assisted profiling and inter-thread communication. Hardware-assisted profiling provides a general mechanism for observing program behavior without diminishing application performance, enabling the VM to take advantage of TLP. However, there must be efficient mechanisms for communicating this information to service threads. Hence, we developed a general memory-based model for communicating profile information.

As noted already, garbage collection provides an excellent opportunity for applying service threads. Profiling hardware is used for observing application stores of object references to the heap, a task traditionally done with inline store barrier code. Then the relevant store information is passed via the memory interface to a service thread that actually executes the store barrier code.

1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 discusses related work in the areas of hardware-assisted profiling, concurrent garbage collection, and multi-threaded optimizations. Section 3 discusses the Relational Profiling Architecture in detail. Section 4 describes the proposed concurrent GC algorithm. Section 5 describes the experimental methods and results, and Section 6 concludes the paper.

2. RELATED WORK

To our knowledge, this is the first paper to use multiple threads and profiling hardware to aid GC. Related work falls into three categories: hardware-assisted profiling, service threads, and concurrent GC.

2.1 Hardware-assisted profiling

The RPA described in Section 3 is most similar to ProfileMe [11]. ProfileMe first picks an instruction for profiling at random. As the instruction flows through the pipeline, information on its behavior is collected and stored in profile registers. When the instruction is completed, whether it is squashed due to a branch misprediction or it is retired, ProfileMe generates an interrupt, and a trap handler reads the information from the profile registers.

ProfileMe reduces hardware by bounding the number of instructions simultaneously profiled, typically to one or two. However, this comes at a cost. ProfileMe lacks some of the expressive power that may be needed. ProfileMe does not provide mechanisms to guarantee particular instructions are profiled, to select which types of instructions are profiled, or to select which information is collected.

The Profile Buffer [10] is a hardware mechanism designed to profile branch edge frequencies in programs. It collects taken/not-taken counts for conditional branches in a buffer following retirement. The buffer is periodically spilled to memory for analysis by software.

Merten et al. [20] develop a scheme for identifying *hot spots* in programs. Hot spots are relatively small regions of static code that account for a large portion of dynamic execution. Their scheme collects branch taken/not-taken counts a structure similar to the Profile Buffer, the Branch Behavior Buffer (BBB). The BBB also identifies frequently seen branches and uses this information to identify hot spots. A separate structure, the Monitor Table prevents hot spot re-detection. RPA can easily collect edge profiles by monitoring the outcomes of executed branches. Tasks such as hot-spot detection would be run on service threads. These threads would maintain structures like the BBB and Monitor Table would be maintained in software.

2.2 Service threads

The service thread concept has been researched under other names for purposes other than GC. Simultaneous Subordinate Microthreads (SSMT) [9] uses multiple *micro-threads* to improve single-threaded application performance. Multiple threads are run along with the application using simultaneous multithreading (SMT) [33][38][37]. Micro-threads are written in an implementation-specific ISA, different from the application ISA, and stored in a special cache structure. Branch prediction is used as an example performance optimization.

Assisted Execution [29] uses *nanothreads* in conjunction with SMT in much the same way. Unlike SSMT and this work, nanothreads share memory and register state with the application thread. Unlike SSMT and similar to this work, nanothreads execute the same ISA as application threads. Nanothreads are invoked either directly by the application, or triggered by hardware events using *nanotraps*. Several data prefetching algorithms are used as example performance enhancements.

2.3 Concurrent garbage collection

Concurrent garbage collection has a long history. Developing an algorithm that is both efficient and correct has proven to be a difficult problem. Sweeping, compared to marking, is easy to perform concurrently with the application. Most research focuses on making the mark phase concurrent. Many of these algorithms were developed as incremental algorithms to reduce pause times on uniprocessors. Usually they could be made completely concurrent, were multiple processors provided.

Several concurrent algorithms are based on Baker's incremental copying algorithm [3]. Before collection, the heap is divided into a *from-space* and *to-space*. From-space contains the objects to be collected, which are copied into to-space by the collector. The collector copies all live objects into to-space. From-space then contains only dead objects and is reclaimed en-mass. The copying process can occur in parallel with the application, as long as an important invariant is maintained. The application may only obtain references to objects in to-space. Whenever the application attempts to load a reference to from-space, the reference is redirected to the copy of the object in to-space, copying the object first if necessary. To accomplish this feat a load-barrier must examine all loads of references.

Assuming from-space is a contiguous region, RPA can perform this check by doing a range check on the loaded value, and throwing a synchronous exception. However, the frequency of such loads is liable to overwhelm the profile mechanism. Hence we avoided algorithms based on Baker's algorithm.

Since maintaining the invariant with software load-barriers is expensive, many systems have proposed hardware mechanisms for performing this check. An early but very impressive GC system was developed for the Symbolics 3600 [21]. This was a system designed explicitly for LISP, and also represents an early co-designed VM. The algorithm is a concurrent copying generational GC algorithm. *Card-marking* [35] was used to track inter-generational pointers. Each card was one virtual memory page.

Tagged memory distinguished references from other data for both hardware and software. Special hardware checked the results of every load instruction. Loading a references to old-space generated an interrupt. The trap handler redirected the loaded word to the copy in copy-space, copying the object first if necessary.

The Symbolics 3600 also had special hardware to track inter-generational pointers. When a reference to a young object (called *ephemeral* in [21]) was stored hardware set a special bit associated with the modified page. These pages were examined for references when the young generation was collected. RPA performs essentially the same function for the collector described here.

The MUSHROOM system [36] used similar forms of support. Tagged memory was used to locate pointers. MUSHROOM, however used a software-controlled object-oriented cache structure. The young generation was maintained in the cache. Objects could be allocated, used, and collected all within the cache, without ever being assigned a physical address. The young-generation collector was not concurrent, although collection of main memory and secondary storage was done incrementally.

Schmidt and Nilsen [27] propose adding hardware support for Baker's algorithm [3] to the memory modules, rather than the CPU. They suggest that adding specialized support in a standard expansion slot will be economically more attractive than modifying the CPU. The *garbage-collected memory module* (GCMM) contains a to-space and from-space, and performs an algorithm similar to Baker's collector. Tagged memory allows the GCMM to identify pointers for the GCMM. The GCMM also traps and updates all reads of references to the from-space, ensuring that the application never obtains references to from-space.

The above hardware mechanisms are designed specifically to support GC and particular languages. Such specific hardware mechanisms have rarely been popular with commercial hardware vendors. The goal of RPA is produce a general mechanism that is useful for a wide variety of tasks. The original intent of RPA was profiling. RPA's ability to perform other services is evidence of its generality.

Doligez et al. developed a concurrent mark-sweep GC algorithm along with a formal proof of correctness [14][15]. Similarly, Lorenz and Winterbottom described a non-generational concurrent mark/sweep collector implemented for the Inferno operating system and the SML/NJ ML compiler [19]. The *very concurrent garbage collector* (VCGC) allows application execution, marking, and sweeping to run concurrently. Essentially, the marker and sweeper are pipelined; the sweeper sweeps what was marked in the previous *epoch*. Objects allocated in the current epoch will be marked in the next epoch, and swept two epochs later.

Like the algorithm we present, both of the above algorithms strive to eliminate as much synchronization as possible. Both of the algorithms handle concurrent reference mutations using a *snapshot-at-beginning* store barrier [35] that observes the reference about to be overwritten by a store. This is a significant difference from our algorithm, which uses an *incremental update* [35] store barrier, observing the stored values, rather than the overwritten values. Observing stored values is easier for RPA, since it need only profile store instructions. RPA can perform a snapshot-at-beginning store barrier by inserting an extra load just before the store instruction, and then profiling that load instruction.

Several concurrent GC algorithms use virtual-memory page protection to replace in-lined barriers. Appel, Ellis and Li [1] use this technique for an algorithm based on Baker's algorithm. Baker's algorithm divides to-space into three areas. The scanned area contains objects that have been scanned for references to from-space. All references in this area have been redirected; the scanned area contains only references to to-space. The unscanned area contains objects that have been copied into to-space, but which have not yet been scanned for references to from-space. Appel, Ellis and Li read- and write-protect the unscanned area. Upon a page protection fault, the handler copies the offending page from to-space into from-space, and updates all the references. Hence the application sees only references to objects in to-space. This algorithm can cause a flurry of page faults at the beginning of GC, due to the root objects being copied into the unscanned area. Such faults are also relatively expensive, since whole pages have to be copied and updated by the handler.

Boehm et al. use this technique to develop a concurrent conservative non-copying generational collector [7]. Copying is avoided because of the focus on conservative collectors. This collector only write-protects pages, and performs only a small amount of work for each protection fault. Faults could be dispensed with altogether, if hardware page dirty bits could be used. Before marking, all of memory is write-protected. For each write-protection fault, the collector notes the modified page and removes the write protection. After marking, objects missed due to concurrent modifications to the heap are detected by following all references from marked (live) objects on modified pages. This second mark can be done without stopping the application if all the pages are re-protected to track modifications. This is done at most twice, at which point the application is stopped and all modified pages are marked one more time with the application stopped. At this point all reachable objects are marked and sweeping can begin. This is similar to how the reference set is handled by our algorithm.

Boehm et al. go beyond proposing a particular algorithm, and propose a general transformation for making a wide variety of non-concurrent GC algorithms concurrent. The algorithm we developed can be considered a variation of this algorithm based on RPA. While they base this transformation on page protection bits, RPA provides the same basic functionality: the ability to track modifications to the heap. Hence, RPA also provides a basis for a wide range of concurrent algorithms, but without the overhead of page faults.

3. THE RELATIONAL PROFILING ARCHITECTURE (RPA)

The RPA is a detailed profiling architecture based on the relational profiling model. This model views profiling as querying a table of profile information. The relational profiling architecture (RPA) is a detailed architecture that embodies the relational model. RPA allows queries to be conveniently expressed and leads to an efficient implementation.

3.1 The Relational Profiling Model

Conceptually, the relational profiling model is similar to a relational database. Dynamic instructions are related to *events*. Events may result from changes to architected or implementation state and *conceptually* can be organized into a table. See Figure 1. This model leads to two basic forms of queries.

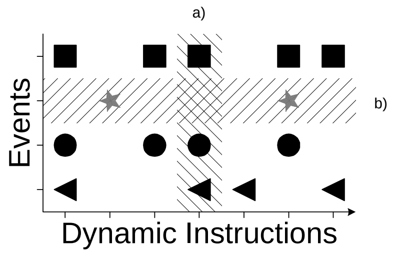

Figure 1. The relational profiling model organizes instructions and events in a table.

1) Instruction-based queries. "For certain instructions, what events occurred?" These queries conceptually select columns from the table. To collect this information, the profile mechanism essentially follows the instruction as it flows through the pipeline, collecting event information regarding its behavior. This is similar to ProfileMe [11].

2) Event-based queries. "For some events, what instructions were involved?" These queries conceptually select rows from the table. To collect this information, the profile mechanism essentially sits at some point(s) in the pipeline, recording information about instructions that flow past. This is similar to the counter profiling mechanisms common in processors today. In contrast to counter-based methods, however, the relational model can provide detailed information regarding specific dynamic instructions. Still, hardware counters may sometimes be used as an efficient summarizing mechanism.

Hybrids are also possible and useful. For instance, the definition of, "some instructions," in instruction-based queries may contain event-related conditions (i.e. "For all load instructions that missed in the cache...").

3.2 RPA Assembly Language

The RPA is most easily understood from its assembly language. Figure 2 shows the RPA assembly language query used for the concurrent garbage collector presented here. The RPA assembler was developed using the ANTLR tool [25] to facilitate research on RPA.

An RPA assembly language statement or "query" 1) describes records of information to be collected, 2) specifies a rate at which the information should be collected, 3) describes selection criteria for which a record should be checked, and 4) indicates actions to be taken for the selected records.

Unlike typical assembly languages, RPA queries invoke a number of machine level instructions that manage various structures in the profile hardware. These structures are described in Subsection 3.3.

```
for opSTORE 1 always collect op1 op2;
        if op1 <> 0 then send 3 stop else stop;
```

Figure 2. Concurrent GC RPA query.

An RPA profile program is broken into a series of queries. Each query begins with a *for* clause, and is followed by one or more *comparison* clauses. The query in Figure 2 contains a single comparison clause.

The *for* clause indicates which instructions should be profiled, what information should be collected and how often. The types of instructions to profile are listed following the *for* keyword. Instructions are divided into the eight classes shown in Table 1. For example, the statement in Figure 2 specifies that store instructions should be profiled.

The VM can further classify instructions using two profile bits per instruction word in the program binary being profiled. This yields a total of 32 instruction classes. Note that the VM paradigm enables an implementation ISA with two embedded profile bits per instruction. An alternative is to add additional hardware tables to hold software-controlled classification information. The two bits are interpreted as an integer. The number following the instruction-class mnemonic indicates what profile bit values should be profiled.

Table 1. Instruction profiling classes.

Mnemonic	Instructions
opJMP	Unconditional jumps
opBRANCH	Conditional branches
opLOAD	Load instructions
opSTORE	Store instructions
opALU	Simple arithmetic and logical instructions
opMULT	Multiply/divide instructions
opFLOAT	Floating point operations
opSYS	SYSCALL, BREAK, etc.

The concurrent GC algorithm only needs to monitor stores that are storing references to the heap. The VM tells RPA which store instructions write references to the heap using the profile bits. Only store instructions with the profile bits set to 1 are profiled, as indicated in the query in Figure 2.

For the specified instruction class, the *for* clause indicates the information to be collected and a random sampling rate. Simple random sampling can reduce the rate at which profile information is collected. Since the GC algorithm relies on observing every reference store for correctness, the *always* keyword indicates that random sampling is not used.

Finally, the *for* clause lists the information collected. Each item represents one 32-bit word of information, which is collected and packed into a record by the RPA. The proposed RPA limits the information collected to seven words per record. This appears to be large enough to handle all profiling tasks without producing overwhelming large records.

Store instructions take three input operands, the value stored (op1), a base address (op2) and an offset. When writing fields in regular objects, the offset is typically a small immediate encoded in the store instruction. When writing arrays, the offset is provided as an input register. In both cases, the base address is a reference to the base of the object being modified. The value stored is the reference being written. The GC query instructs RPA to collect the reference being stored and the reference of the object being modified to monitor concurrent modifications to the heap during marking, and to track inter-generational pointers.

Following the *for* clause, *comparison clauses* indicate conditions used to select certain records and the actions to be taken for the selected records. Each comparison clause can perform up to two comparisons to check for desired properties within the record. The comparison types available include the standard relational operations, comparisons that check for set or cleared bits, and further random sampling. If the comparison(s) match, an action may be performed and/or a branch to another comparison clause may be taken Otherwise, execution falls through to the next sequential query instruction. The *stop* keyword within the comparison clause indicates that query is completed. Since stores of null references do not need to be examined by the GC algorithm, the query in Figure 2 checks if operand 1, the stored reference, is not null (zero).

Profile actions communicate collected profile information back to VM software. The most common action is the *message send*, indicated by the *send* keyword, as shown in Figure 2. A copy of the record is written into a circular message queue where it can be examined by a service thread. The service thread can then perform more detailed processing of the information. The query in Figure 2 sends the collected record to a service thread that performs the store barrier function. Message queues are held in shared memory, and service threads access these queues using normal loads and stores. The RPA can also disburse messages can be disbursed to messages to multiple queues. The RPA will perform load balancing among the queues so multiple service threads can efficiently process the records in parallel. Collections of queues used in this manor are called a *pool*. Hence RPA send actions provide an automatic mechanism for single-producer/multiple-consumer communication. Furthermore, the query engine can send records with different information to different pools. This way service threads only receive information related to their task, eliminating the need to determine what type of message has been received. The number following the *send* keyword specifies the queue pool to which the record is to be written.

3.3 Low-Level Architecture

Figure 3 illustrates an implementation of the RPA. Specific queries are formed by the virtual machine using the query language described in the preceding section. Using this description the assembler divides query processing into two components shown in the figure. A configuration for the *profile control table* (PCT) is derived from the *for* clauses. The PCT is a set of architected *profile control registers* (PCRs). Software configures the PCT using a special SET_PCR instruction added to the I-ISA.

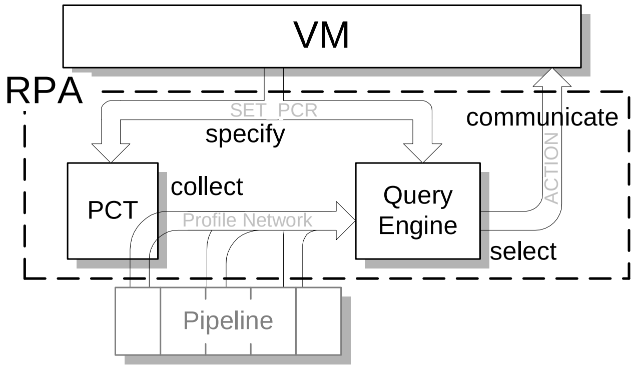

Figure 3. The relational profiling architecture contains the profile control table (PCT) and the query engine.

Comparison clauses generate query instructions to be executed by the *query engine*, a simple processor capable of performing the comparisons and actions dictated by the query. Query instructions may be stored in memory, or in a special-purpose table constructed out of PCRs to reduce implementation costs. A special-purpose table is currently simulated.

3.3.1 The Profile Control Table (PCT)

The PCT implements two PCRs for each of the 32 instruction classes. The first PCR sets the sampling rate, and selects information to be collected. The second PCR contains the starting *query PC* (QPC), the address of the initial comparison instruction for the query engine to execute. Profile records collected at the behest of the PCT are passed to the query engine for further processing.

3.3.2 The Query Engine

The query engine begins executing the comparison instructions at the initial QPC location provided by the PCT. Comparison instructions are executed until terminated by an explicit stop annotation within an instruction. The application program continues to execute in parallel with the query engine, and multiple comparisons may be executed in parallel as well. To simplify implementation there is no guarantee of the order in which separate queries are executed or completed.

Each comparison instruction can specify up to two comparisons, a branch and a profile action. The query engine can compute arbitrary Boolean expressions by forming comparison instructions into if-then-else decision trees.

The comparison instructions are encoded in a 64-bit format. The basic two-comparison format is shown in Figure 4. The comparison instruction reads two values from the record. Comparisons operate on 8,16 or 32 bit data, so both the size and location of the data to examine are encoded in the instruction. To reduce implementation costs, one even word and one odd word from the profile record are selected. The comparison instruction also includes a 16 bit immediate value.

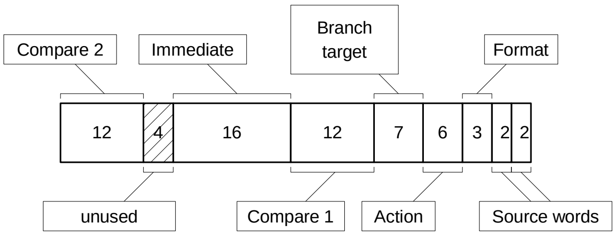

Figure 4. Comparison instructions contain up to two comparisons, a branch and an action.

If both comparisons in the instruction match, the specified action is performed and the branch is taken. Otherwise execution falls through to the next sequential comparison instruction.

To implement message actions, the query engine manages circular message queues in memory. Each message uses eight words of memory, and the size of the queue is configurable up to 128 messages. To write a record into a queue, the query engine writes the seven words of the record. The eighth word is used as a ready word to indicate that the record is available. To read a record, a service thread polls the ready word, reads the record from memory, and then clears the ready word. The service thread also periodically informs the query engine how many messages have been read. This is done by storing the total messages read into the *buffer read status* word in memory, which is examined periodically by the query engine. The query engine uses the number of messages read, along with the number written (which it knows), to determine if space is available for another message. Using the buffer read status word reduces the polling required by the query engine, over using the ready words alone.

3.4 RPA Implementation and Cost

The query engine pipeline modeled for this research is shown in Figure 5. The query engine is designed to execute up to four queries simultaneously using a simple barrel-and-slot design [31]. The pipeline is four stages long, and executes one query instruction for each active query once every four cycles in a round-robin fashion. As the profile records fill, the processing power of the query engine increases to one query instruction per cycle. The barrel-and-slot design eliminates all interlocks and dependences in the pipeline as well as branch misprediction penalties.

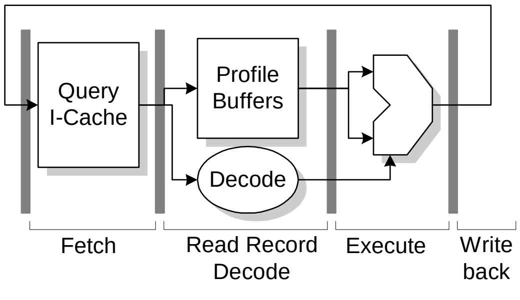

Figure 5. The query engine four stage pipeline.

Messages are sent by the *message engine*. The message engine reads records from the record buffer using a second dedicated port, and writes them to the in-memory queues. The message engine also handles polling the buffer read status words.

A profile network is needed to carry profile information from the pipeline to the profile buffers. Though a complete design of this network is beyond the scope of this paper, the size of the network will scale linearly with the number of simultaneously profiled instructions [11]. The latency of this network is not a concern; the profile mechanism can be relatively distant from the core pipeline. However the size and layout impact of this network on the core should be minimized.

Instructions are selected for profiling during instruction dispatch. At this point a profile buffer is allocated to store the collected information until the query engine finishes processing it, and a profile network is allocated to carry information for the profiled instruction. If either is not available then dispatch stalls. When the instruction retires the profile network is freed, but the profile buffer remains allocated until the query completes. Four profile networks and eight profile buffers, 256 bytes, is enough to make stalls due to profiling a rarity, as will be shown in Section 5.4.

4. CONCURRENT GARBAGE COLLECTION USING THE RELATIONAL PROFILING ARCHITECTURE

Concurrent [19][14][32][35] and generational [34] garbage collection schemes must monitor stores to the heap while collection is in progress. This *store barrier* typically takes the form of several instructions inserted before every instruction that stores a reference to the heap.

As shown in Figure 2 and described in Subsection 3.2, the RPA replaces inlined store-barriers by profiling the store instructions. The RPA collects the needed information, and sends this information to store barrier threads, which execute the store barrier in parallel with the application. Three store barrier threads are currently used. This is enough to keep up with application stores, though it may be possible to use fewer threads.

Furthermore, the store barrier changes slightly depending on whether a collection is actually taking place. Dynamically altering store barrier behavior is easy with RPA, unlike traditional inlined store barriers. The store barriers simply check a global variable to see which type of store barrier needs to be executed. This check is only performed when the message queues are empty, so the overhead of the check occurs only when there is no other work to do.

The resulting GC algorithm is an almost entirely concurrent two-generation algorithm. Because moving objects concurrently is difficult [14], we focused on non-copying generational collection [7][12]. The application must stop for three short pauses during garbage collection, as is explained later.

4.1 Heap layout

We organize the heap in a manner similar to the "big-bag-of-pages" organization used in the Boehm-Weiser conservative collector [6]. Heap space is allocated in chunks, and each chunk contains an integer number of same-size blocks. Chunks are nominally 1KB. However, unlike many collectors using this layout, all chunks need not be the same size, and they only need be double word aligned. If blocks of a given size do not fit evenly into a chunk, slightly smaller chunks are allocated for that size.

The heap is divided into the young generation and the old generation. Each generation is composed of a doubly-linked list of chunks. Chunks are moved between generations by removing them from one list and inserting them into the other.

Supporting all block sizes is inefficient, as there are frequently not enough objects of a given size to fill up a chunk. This is especially true for larger sizes. All block sizes divisible by eight bytes are supported up to 128 bytes; block sizes divisible by 16 bytes are supported up to 256 bytes. This combination appears to yield good performance. An object will use the smallest size block into which it fits. Objects larger than 256 bytes are allocated separately.

The first word of each object, called the *method table pointer* (MTP), points to type information for the object. This is used for virtual function calls, type checks, and to find reference masks used to locate the references within the object. The GC algorithm uses the lower three bits of this pointer for GC state. These bits must be masked off before the MTP can be used. This overhead has not been found to be significant. If this overhead should be an issue, a load-and-mask operation could be added to the I-ISA. The advantage of this, versus storing the bits off to the side, is that the bits can be easily found given the object reference or the block address. Storing the state elsewhere generally results in a relatively complex instruction sequence just to locate the state.

The three MTP bits are the *OLDGEN_MARK* set for objects in the old generation, the *REM_MARK* used to maintain the *remembered set* described in the next subsection, and the *LIVE_MARK* set when an object has been reached by the marking process.

Marking an object involves setting the LIVE_MARK, and placing the object on the *mark stack*. Objects are popped off the mark stack, and scanned for references to unmarked objects, which do not have the LIVE_MARK set. This process iterates until the mark stack empties, at which point all reachable objects have been marked. During sweeping, objects with the LIVE_MARK unset are collected and the LIVE_MARKs are reset.

Unallocated blocks are placed in per-size singly linked lists stored in the free blocks themselves. Object allocation is inlined in the application code, and involves popping a free block off the list. Except for the first two words of memory, used for the free list, the GC algorithm initializes memory to zero concurrently, so allocation results in an initialized block.

If no free block is available then a new chunk must be allocated and initialized. This more complex operation is performed through a call to the run-time system. If this expands the heap beyond a dynamically adjusted threshold, GC is invoked. Typically only the young generation needs to be collected. If tenuring expands the old generation beyond a dynamically adjusted threshold, the entire heap is collected.

Whole chunks are tenured by unlinking them from the young generation and linking them into the old generation. In addition OLDGEN_MARK is set when an object is tenured into the old generation. GC uses OLDGEN_MARK to avoid marking old generation objects when collecting only the young generation.

Tenuring free blocks wastes space since new objects cannot be allocated into the old generation. The tenuring policy tries to minimize this waste by selecting nearly full chunks. Only chunks that are at least 75% full are tenured. In practice, between 0% and 14% of the tenured blocks are free. In addition, chunks are de-tenured when they become mostly free. In order to reclaim otherwise wasted space, blocks are de-tenured when no more than 25% full. Demers et al. describe a similar solution for their non-copying conservative generational collectors [12]. This mixes old-generation objects into the young generation. Sweeping the young generation becomes slightly less efficient, because old objects must be explicitly skipped over. However, in practice less than 2% of objects in the young generation are old-generation objects.

The GC algorithm uses load-locked/store-conditional primitives to atomically examine and set MTP bits. However, the application does not require *any* synchronization with the GC algorithm. Since the application does not modify the MTP or use the GC state bits, there are no synchronization issues with these bits.

The algorithm also eliminates all synchronization hazards between allocation, which pops objects off the free lists, and sweeping, which places collected object on the free lists. The young generation is divided into two portions. New objects are allocated by the application into the *crib*. GC sweeps the *sweep area*. Before sweeping, the application is paused temporarily, and all objects are (logically) moved from the crib into the sweep area, leaving the crib empty. After sweeping, the application is again paused temporarily, and uncollected objects are merged back into the crib. Fine-grain synchronization is unnecessary because collected objects are placed en mass on the free-lists while the application is paused. This also prevents objects that were allocated during sweeping from being erroneously swept and collected. A similar optimization is performed in VCGC [19].

4.2 Remembered set and inter-generational references

To collect only the young generation, some mechanism must track references from the old generation into the young generation. This is done using the REM_MARK MTP bit and the remembered set. The remembered set is a list of all objects in the old generation that may contain references to objects in the young generation. REM_MARK is set for all objects in the remembered set to avoid adding duplicate entries to the remembered set.

Store-barrier threads examine the REM_MARK of modified objects, as reported by RPA. If it is not set, the object is placed in the remembered set, and the REM_MARK is set. To prevent young-generation objects from being placed in the remembered set all objects in the young generation have the REM_MARK set.

The resulting extremely efficient store barrier is shown in pseudo-code in Figure 6. Although this is shown in C-like code, each statement represents one machine instruction. For most applications only a few old-generation objects point to young-generation objects, so only a few objects need be placed in the remembered set. In benchmarks we have observed REM_MARK is already set 99.99% of the time. Hence, the basic store barrier is 3 instructions in the common case. The resulting remembered set is free of duplicates, and contains exactly those objects in the old generation that must be examined.

```
Store_bar:
  mtp = load_locked(obj.mtp);
  rem_mark = mtp | REM_MARK;
  If (mtp != rem_mark) {
     if (!store_cond(obj.mtp, rem_mark)) goto Store_bar;
     // Add to remembered set …
  }
```

Figure 6. Generational store barrier pseudo-code.

Objects referenced by the *root references*, references in local and global variables, are marked first. When collecting the young generation, young objects referenced from old objects in the remembered set are also marked. Several other marking steps are required to make the algorithm concurrent, as explained in the next subsection.

4.3 Concurrent modifications of the heap

Concurrent collection also requires the store barrier to keep track of modifications to heap references during marking. Otherwise, GC could collect live objects when the application moves references during marking. A simple example is shown in Figure 3. In Step 1 GC has marked and scanned object A, placing object B on the mark stack. However in Step 2, before the GC scans object B, the application moves object B's reference to object C into object A. Finally in Step 3, GC scans Object B. Object C is missed, and is mistakenly collected.

To solve this problem, we use an *incremental update* [35] scheme. Whenever a reference to an object is written to the heap, the referenced object is marked live, and placed in the *reference set*. This set contains all the objects that might otherwise be missed due to application modifications of the heap.

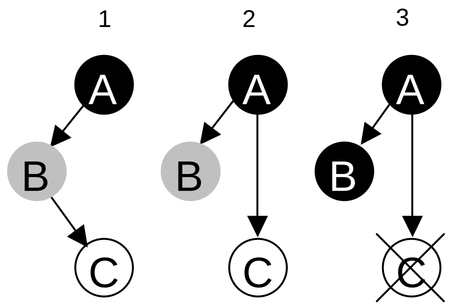

Figure 7. Example concurrent reference mutation.

We use a store barrier mechanism similar to the one used for the remembered set. RPA is used to collect the same information from every reference store, the stored reference and a reference to the modified object. If the LIVE_MARK of the referenced object is not set, then LIVE_MARK is set and the object is placed on the reference set. In our benchmarks, at least 98.5% of the time the LIVE_MARK is already set, and no more work needs to be done. The query engine simplifies the concurrent store barrier code by doing the required null reference check.

The store barrier thread must execute two or three additional instructions to remove the profile message from the message queue. This results in a five-instruction store barrier in generational mode when not marking concurrently, and a twelve-instruction store barrier in concurrent mode during marking.

4.4 Complete algorithm

Figure 8 provides a time line of the algorithm's execution. Time runs down the image, and is not to scale. A collection of the young generation is depicted and described. Collecting the entire heap is similar but simpler. Five threads are shown, the application thread, three identical store barrier threads, and the garbage collection thread, which executes the marks and sweeps. The wide crosshatched bars indicate where the application is running. The narrow vertical lines indicate where the GC algorithm is running. No line indicates that the thread is paused. Horizontal dashed lines depict inter-thread control

The complete algorithm works as follows:

Steps 1-4. The application tries to expand the heap, exceeding the GC threshold. This will begin the GC process. Next the application thread builds the set of root references, and switches the store barrier from the generational store barrier to the concurrent store barrier. Then, GC is started and the application continues execution.

Step 5. The GC thread marks the root set built in Step 2

Step 6. All objects in the remembered set are scanned for references to young-generation objects. These young-generation objects are marked. To prevent the store-barrier threads from modifying the remembered set being examined by the GC thread, the GC thread atomically snapshots the current state of the remembered set. Store barrier threads find the remembered set using a global pointer within the run-time system. To snapshot the remembered set, the GC thread grabs a copy of the global pointer, and swaps in a pointer to a spare remembered set. Mutual exclusion locks make this operation atomic. During the rest of Step 6 store-barrier threads will add objects to the spare remembered set. At the end of Step 6 the original and spare remembered sets are merged.

Objects remembered during this phase will not be examined during this collection. This is not a problem because the concurrent store barrier will add any referenced young objects to the reference set, which is repeatedly marked in the next step.

Objects are removed from the remembered set if no references to young objects are found. This creates a subtle synchronization hazard, which must be guarded against. A scenario is illustrated in Figure 9.

In Step 1 of Figure 9 a remembered object is being scanned from beginning to end for references to young-generation objects. In Step 2, the application stores a reference to a young object into the remembered object. However, the GC thread has already moved past this reference field and does not see the new reference. In Step 3, the GC thread, believing the object contains no references to young-generation objects, erroneously removes the object from the remembered set. The referenced young object will *not* be collected during this GC, because the store barrier will add it reference set in Step 2 of Figure 9. It will be erroneously collected in the *next* GC, if the stored reference is the only reference to the object. Because the old-generation object is not in the remembered set, the collector will not find or follow the reference.

Clearing the REM_MARK before scanning remembered objects solves the problem. In the scenario just described the store barrier thread will add the object to the spare remembered set, and set the REM_MARK again. To protect against duplicate entries in the

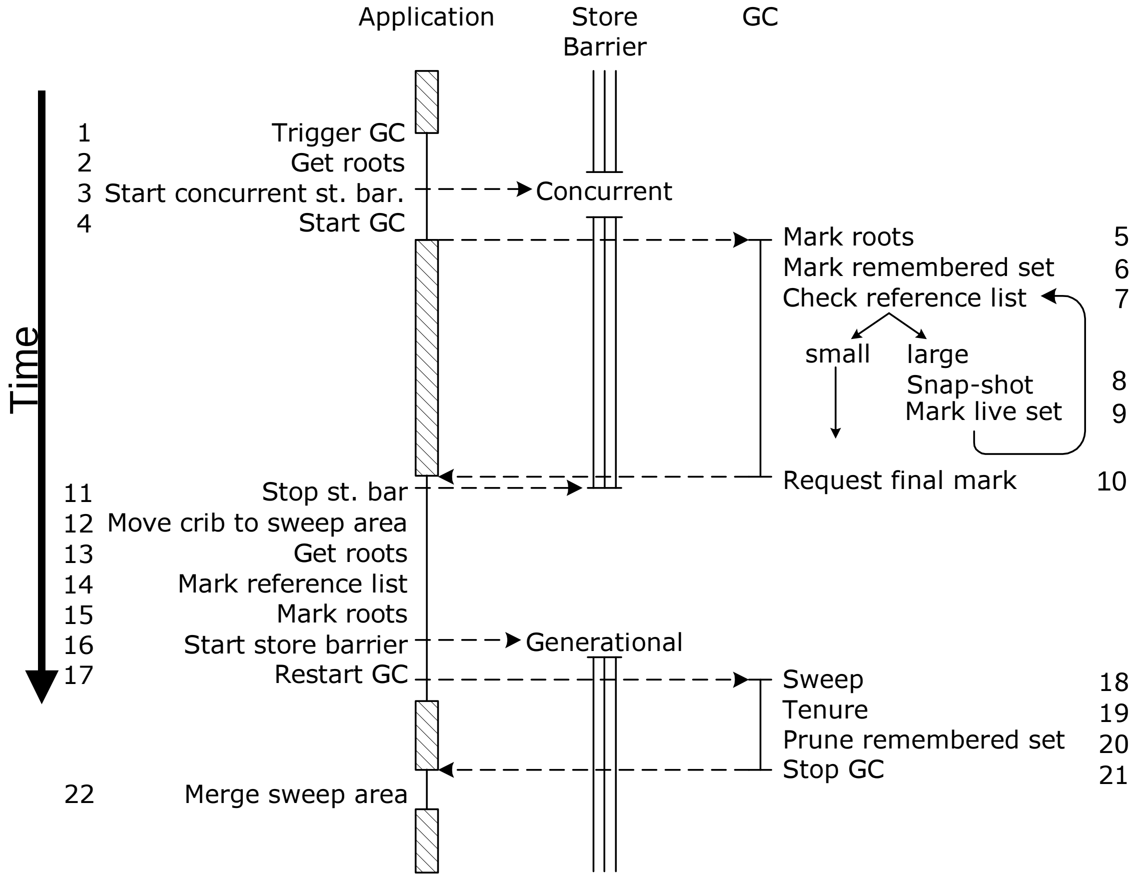

Figure 8. Concurrent GC algorithm

remembered set, the GC thread examines the REM_MARK again after scanning an object if young-object references were found. If it is set, the object is removed from the snapshot copy of the remembered set, since a duplicate was added to the spare set.

Steps 7-9. Eventually the application must be stopped, and some amount of marking must be done with the program paused. The goal of a good algorithm is to minimize the duration of this pause. The size of the reference set, built as explained above by the concurrent store barrier, is examined. If the reference set is large, it will be repeatedly marked in parallel. If it is small, it will be marked while the application is paused, and marking will terminate. The threshold between "small" and "large" is doubled each iteration, guaranteeing the loop will terminate.

The size threshold starts at 20 objects, and is approximately doubled every iteration. The loop exits when the number of objects on the reference set is less than the threshold. In addition, we found it was very important to do at least one parallel mark, even if there are only a few entries in the initial reference set. The number of entries only approximately indicates the amount of work to be done during marking, because a single reference could point to a large body of unmarked objects. This appears to happen quite regularly for the initial reference set, so it is important to mark the initial reference set in parallel regardless of its size. While the algorithm makes no guarantees, this does not seem to occur during subsequent passes through the loop. For subsequent iterations the number of reference set entries is an accurate measure of the time needed to mark them.

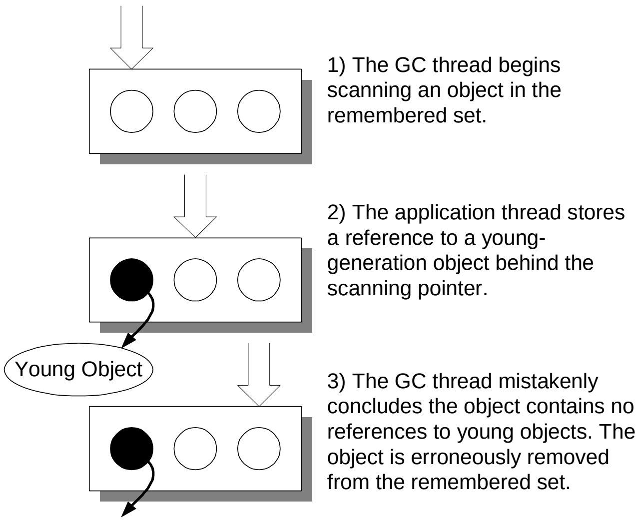

Figure 9. Removing objects from the remembered set creates a subtle synchronization hazard.

When marking the reference set in parallel, the GC thread atomically takes a snapshot of the current reference set to prevent the store barrier from modifying the reference set that the GC thread is examining.

Steps 10-17. Once a small reference set has been obtained, it is time for the final serial mark. The application stops and pauses the store barriers. This process ensures that there are no unexamined store-barrier messages in the system. At this point the application thread (logically) moves objects from the crib into the young generation, leaving the crib empty, as explained in Subsection 4.1.

The application can then mark the final reference set, and obtain and mark the final root set. At this point, all live objects have been marked. The store barrier threads are restarted in generational-only mode, and the application restarts while the GC thread sweeps the sweep area in parallel.

Steps 18-21. Dead objects in the sweep area of the young generation are collected. When collecting the entire heap, objects in the remembered set may be freed, and these entries must be removed from the remembered set. This is done with an explicit scan of the remembered set following sweeping. Finally, GC is complete. The GC thread informs the application that it is finished, and stops.

Step 22. The application, when preparing to allocate more memory, notices that GC has finished. At this point it (logically) merges the sweep area back into the crib, allowing recently freed objects to be reused. The sweep area is left empty for the next GC.

4.5 Correctness

GC correctness, defined as never collecting reachable objects, is difficult to prove in the presence of concurrent modifications by the application. Although this has been done formally for some algorithms [14], we must leave this for future work.

The application complicates GC when it moves a reference from one location to another. Moving a reference involves first making a copy of it, and, second, overwriting the original copy. Neither copying nor overwriting alone is sufficient to undermine the GC process. Copying alone is not a problem; the original reference still exists to be traced during marking. Overwriting alone is not a problem; if the reference is actually gone, it does not need to be followed.

This algorithm assures correctness by monitoring the copied references. Other algorithms, as described in the Subsection 2.3 on related GC algorithms, monitor overwritten references. The goal of this algorithm is to assure that a copied reference will be traced during marking, no matter where it is copied. There are three regions to which a reference may be copied: local and global variables, the young generation and the old generation. If the reference is copied to local or global variables, it will be caught when the roots are obtained the second time in Step 13, and marked in Step 15. If the reference is copied to the young generation, it will be placed on the reference list by the store barrier threads, and marked in either Step 9 or 14. The same is true for copies made to the old generation. In addition, the object in the old generation is placed in the remembered set, guaranteeing that the copied reference will be marked in future GC passes as well.

5. EXPERIMENTAL METHODOLOGY

Our experiments are based on the Strata VM for Java and the SimpleMP simulator [26]. SimpleMP is a version of the SimpleScalar [8] execution-driven timing simulator of out-of-order processors that was extended to simulate multiple processors. This was further extended to simulate SMT and RPA.

5.1 The Strata compiler

Co-designed VMs use a combination of interpretation and dynamic compilation to transparently execute programs encoded in the V-ISA on hardware directly supporting the I-ISA. This research focuses on the interactions between compiler, run-time system, architecture and micro-architecture. Rather than creating a complete VM, Strata statically compiles Java bytecodes to SimpleScalar PISA assembly code. To aid in development, Strata also targets SPARC assembly. This provides a simple and flexible system for research, while reducing the effort required for building and maintaining the system. Essentially, only the execution of the run-time system (including GC), and dynamically optimized code is simulated. Other phases of execution, such as compilation and interpretation, are not simulated. In a well tuned VM, these phases of execution should not dominate execution time. We expect future VMs to perform compilation in parallel with application execution (using service threads), and possibly to preserve compiled code across executions of the same program.

The Strata compiler is itself written in Java, and forms one of our better benchmarks. The Strata compiler performs typical optimizations such as global register allocation, constant propagation, local common sub-expression elimination and global copy propagation. It also performs Java-specific optimizations aimed at eliminating null-pointer checks, type checks and array bounds checks.

Strata is still under development, and does not perform some common optimizations. In the future we plan to include forms of global code motion, such as partial-redundancy elimination (PRE) [22], and function inlining. These optimizations will make low-overhead GC more important, since the application will be executing relatively faster.

The runtime system, which contains the GC algorithm, is written in C. Running a Java application involves compiling the bytecodes using the Strata compiler, and then linking the resulting assembly with the runtime system.

5.2 Benchmarks

The four benchmarks shown in Table 1 are simulated. The first is the Strata compiler. The other three are taken from the SpecJVM98 suite. They are DB, a simple relational database, Jack, a parser generator, and Ray, a 3-D rendering tool. Applications are simulated from about 10 million instructions before the first GC, to completion.

Table 2. Benchmark characteristics.

Bench-mark	Input Data	Duration (Millions) Instr.	Cyc.	GCs
Strata	spec.benchmarks. _209_db.Database	557	228	6
Ray	50 500 time-test.model	282	79	3
DB	db2 src3	241	78	1
Jack	10	692	244	7

5.3 Processor models

Ways to exploit TLP include multiple processor cores and simultaneous multithreading, which allows instructions from multiple threads to flow through the pipeline at the same time. Service threads that access much of the same data as the application should be executed on the same processor core using SMT to prevent data from being bounced between processor caches. Threads using different data should be executed on different cores to prevent L1 cache pollution. Since GC generally has poor locality characteristics, we chose to execute the GC algorithm on separate cores. Quantitatively analyzing different design points is on-going research.

The particular design explored in this paper, Figure 10, uses both SMT and multiple on-chip cores. On one chip, there is a large high-ILP processor supplemented by two *service processors*. The computation of greatest concern, the application program, runs on the high-ILP processor. Given many service threads, more instruction throughput can be obtained merely by stamping out more processors. Hence, the service processors are designed to maximize instruction throughput per unit area, rather than single-thread performance at any cost. Lower priority VM tasks, in this case GC, run on these processors concurrently with application execution.

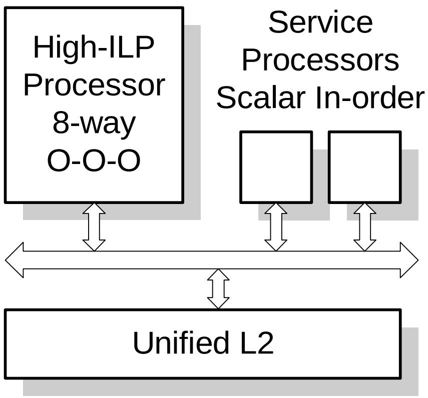

Figure 10. System-on-a-chip design.

Parameters for the processor models are shown in Table 3. The ILP processor is an 8-way superscalar, out-of-order processor, running only one thread. Each service processor is a six stage scalar pipeline capable of running 3 threads in an SMT fashion. To keep these processors small and simple, they have small L1 caches and predict-not-taken branch prediction. All three processors connect to an L2 cache with a 12-cycle access time. The L2 is perfect; it never misses.

Table 3. Processor model parameters.

Parameter	High-ILP Processor	Service Processor	Units
Processors	1	2	Proc.
Threads	1	3	Threads
Width	8	1	Instr.
Instr. Win.	128	(in-order)	Instr.
Br. Pred.	8K entry gshare	Not-taken	
Min. penalty	8	4	Cycles
I-Cache	32KB 2-way	1KB 4-way	
D-Cache	64KB 4-way	2KB 4-way	
Unified L2	Perfect		

5.4 Results

Table 4 shows simulated times for various phases of GC, averaged over all GCs in each benchmark. The first three columns of data show the pause times for the application thread. Referring back to Figure 4, the first pause is for initial collection of the roots, Steps 1-4. The second pause is for the final root collection and mark phase, Steps 11-17. The third pause is for merging the collected sweep area back into the crib area of the young generation, Step 22. These pause times represent one of small remaining sources of GC overhead. Times are shown in 1000's of cycles. For a 1GHz clock, this corresponds to microseconds. The second pause, involving both a root collection and a small mark, is always the longest pause. The largest pause time of 53,000 cycles was observed for the Strata benchmark.

Column 4 shows the cycles (in thousands) required to mark the root set (Figure 4, Step 5). Column 5 shows the cycles required to mark the remembered set (Step 6). During the first collection the old generation and the remembered set are both empty. As shown in Table 2 DB only performs one collection, and so spends little time on the remembered set.

Columns 6 through 8 give statistics indicating the performance of the iterative reference set marking process, Steps 7 through 9 in Figure 4. Column 6 (Live) shows the total time for this loop, Column 7 (N) indicates the number of iterations, and Column 8 (Final) shows the number of objects in the reference set when the loop exited. DB spends more time marking the reference set than other benchmarks because the reference set grows to over 2000 entries. For all other observed collections the number of entries ranged between 0 and about 1000. Marking was never performed in parallel more than once because of the small number of additional entries added to the reference set. For Raytrace some collections the reference set was empty, which led to an average number of iterations less than one.

Finally, columns 9 through 11 give times in cycles for sweeping, tenuring and pruning. Tenuring and pruning take a relatively short time. Sweeping is the longest single collection phase as shown in Figure 11.

Table 4. GC Performance Characteristics (Thousands of cycles)

Benchmark	P1 (1)	P2 (2)	P3 (3)	Root1 (4)	Rem Set (5)	Live (6)	N (7)	Final (8)	Sweep (9)	Tenure (10)	Prune (11)
Strata	33	42	28	2844	5708	1744	1.0	4	13165	926	1812
Ray	19	33	25	316	4195	1281	0.7	7	11010	372	340
DB	15	19	6	1970	1	13443	1.0	0	16888	380	306
Jack	23	29	21	1702	4084	196	1.0	0	11890	534	703

Figure 11 represents similar information graphically for the second garbage collection in the Strata benchmark. The total duration of the collection is a little less than 20 million cycles, shown on the X-axis. The upper time line shows the various GC phases, and the lower line shows the application pauses. The pauses are so small that the hash marks plot on top of each other. The sweep phase is easily the longest phase. This involves scanning the heap for dead objects, zeroing freed object, and putting them on a free list for reallocation.

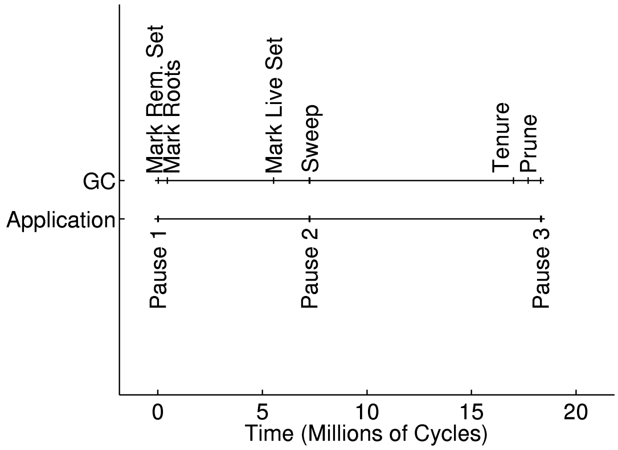

Figure 11. Time line for the second GC in the Strata benchmark.

Table 5 describes the time overhead of GC for each benchmark. The first column of data shows the percentage of cycles that dispatch was stalled due to a lack of profile resources. Simulation results not shown indicate that these stalls occur due to limited profiling networks, not limited profile buffers. Jack stores references much more frequently than the other benchmarks, stressing the RPA. The out-of-order execution window covers about half of these stall cycles. The second column shows the actual slow down compared to an implementation with unlimited profile networks and buffers. The third column shows the percentage of cycles spent in the three pause times. The final column sums the previous two, giving the total time overhead for GC. Jack experiences the greatest GC overhead of 1.32%. Average overhead is a miniscule 0.55%. Direct comparisons to previous collectors are difficult, due to varying languages, benchmarks and VM implementations. However, GC overhead, as well as the overhead for explicit memory management in C and C++ programs, is typically reported as being 20% or more [2][24][13][40][39].

Table 5. Garbage collection time overhead.

Bench-mark	Profile Stalls (%)	Profile Overhead (%)	GC Pauses (%)	Total GC Time Overhead (%)
strata	0.46	0.18	0.31	0.49
ray	0	0	0.34	0.34
db	0	0	0.05	0.05
jack	2.06	1.11	0.21	1.32
Avg.	0.63	0.32	0.23	0.55

Cache interference is another source of overhead not shown Table 5. GC can interact with the application through the memory hierarchy, slowing the application down by causing extra cache misses. Future research will examine these effects.

Table 6 shows the effectiveness of null-reference check performed by RPA. The first column of data shows the cycles per profiled store instruction. Store-barrier threads would be completely overwhelmed by the Jack benchmark if the RPA did not filter stores of null references. The percentage of records eliminated through this check, shown in the second column, varies widely across the benchmarks. Jack stores a null reference 95% of the time. This does not include instructions that store null as an immediate value. Jack makes frequent use of the hash table in the Java library. Almost all of the null reference stores in Jack occur as a result of one store instruction within that library. The final column shows the number of cycles per message handled by the store barrier threads. The amount of work performed by the store-barrier threads varies by a factor of two, suggesting that the VM adapt the number of threads assigned to this task to the workload.

Table 6. Store barrier work eliminated by RPA.

Bench-mark	Cycles per Profiled Store	Null Reference Stores (%)	Cycles per Processed Store.
strata	102.3	28.2	142.6
ray	390.4	0.0	390.5
db	142.2	0.0	142.2
jack	16.3	94.9	319.6

6. CONCLUSIONS AND FUTURE RESEARCH

The Relational Profiling Architecture provides a low-overhead flexible mechanism for inter-thread communication. It allowed the concurrent GC algorithm to monitor stores with very little overhead, provided a more flexible store barrier that could change depending on needs of the moment, and filtered out unnecessary null-reference stores. This enabled the Strata virtual machine to take advantage of low-cost thread-level parallelism, resulting in a GC system with pause times of 53 microseconds or less with a 1GHz clock cycle, and an average time overhead less than 0.6%.

Further improvements can be made to the GC algorithm. All phases of the GC algorithm can be highly parallelized. Parallelizing these phases will become necessary as Strata moves on to multi-threaded applications in order for the GC algorithm to keep up with multiple application threads.

The RPA is a low-cost general mechanism, enabling such improvements for a wide range of GC algorithms, as well as a host of other profiling tasks. Future work includes using the RPA to collect profile information for such optimizations as function inlining, basic-block ordering, and partial-redundancy elimination [22].

7. ACKNOWLEDGEMENTS

This work was support by NSF Grant CCR-9900610, by Sun Microsystems, by an IBM Partnership Award, and by Intel Corporation. The authors would also like to thank the reviewers for their helpful comments, as well as Mario Wolczko and Subramanya Sastry.

8. REFERENCES

[1] Andrew W. Appel, John R. Ellis, Kai Li, "Real-Time Concurrent Collection on Stock Multiprocessors," 1988 Conf. on Programming Language Design and Implementation, pp. 11-20, June 1988.

[2] Eric Armstrong, "Hotspot, A New Breed of Virtual Machine", JavaWorld, March 1998.

[3] H. G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM, 21(4), pp. 280-294, April 1978.

[4] Luiz André Barroso, et al., “Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,” 27th Intl. Symp. on Computer Architecture, pp. 282-293, June 2000.

[5] Viktors Berstis, "Security and Protection of Data in the IBM System/38," 7th Intl. Symp. on Computer Architecture, pp. 245-252, 1980.

[6] H. Boehm, M. Weiser, "Garbage Collection in an Uncooperative Environment," Software Practice and Experience, pp. 807-820, Sept. 1988.

[7] Hans J. Boehm, Alan Demers, Scott Shenker, “Mostly Parallel Garbage Collection,” SIGPLAN Conf. on Programming Languages Design and Implementation, pp. 257-264, June 1991.

[8] Douglas C. Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin - Madison Comp. Sci. Tech. Report #1342, June 1997.

[9] Robert S. Chappel, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, Yale N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," 26th Intl. Conf. on Computer Architecture, pp. 186-195, May 1999.

[10] Thomas M. Conte, Kishore N. Menezes, Mary Ann Hirsch, "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer," 29th Intl. Symp. on Microarchitecture, pp. 36-45, Dec. 1996.

[11] Jeffrey Dean, James E. Hicks, Carl A Waldspurger, William E. Weihl, George Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," 30th Intl. Symp. on Microarchitecture, pp. 292-302, Dec. 1997.

[12] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, Scott Shenker, “Combining Generational and Conservative Garbage Collection: Framework and Implementations,” 17th Symp. on Principles of Programming Languages, pp. 261-269, Jan. 1990.

[13] David Detlefs, Al Dosser, Benjamin Zorn, "Memory Allocation Costs in Large C and C++ Programs," Univ. of Colorado at Boulder Tech. Rep. #CU-CS-665-93, 1993.

[14] Damien Doligez, Xavier Leroy, "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML," 20th ACM Symp. on Principles of Programming Languages, pp. 113-123, Jan. 1993.

[15] Damien Doligez, Georges Gonthier, "Portable, Unobtrusive Garbage Collection for Multiprocessor Systems," 24th Symp. on Principles of Programming Languages, pp. 70-83, Jan. 1994.

[16] Kemal Ebcioglu, Erik R. Altman, "DAISY: Dynamic Compilation for 100% Architecture Compatibility," IBM Research Report RC 20538, Aug. 1996.

[17] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, David Appenzeller, "Dynamic and Transparent Binary Translation," Computer, pp. 54-59, Mar. 2000.

[18] Alexander Klaiber, "The Technology Behind Crusoe Processors," a Transmeta technical brief, 2000.

[19] Lorenz Huelsbergen, Phil Winterbottom, "Very Concurrent Mark-&-Sweep Garbage Collection Without Fine-Grain Synchronization," Intl. Symp. on Memory Management, pp. 166 - 175, 1998.

[20] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, Wen-mei W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," 26th Intl. Symp. on Computer Architecture, pp. 136-147, May, 1999.

[21] David A. Moon, "Garbage Collection in a Large Lisp System," 1984 Symp. on LISP and Functional Programming, pp. 235-246, Aug. 1984.

[22] E. Morel, C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," Communications of the ACM:22(2), pp. 96-103, 1979.

[23] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K.-Y. Chang, “The Case for a Single-Chip Multiprocessor,” 7th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems, Oct. 1996.

[24] James O'Toole, Scott Nettles, "Concurrent Replicating Garbage Collection," Proc. of the 1994 ACM Conf. on LISP and Functional Programming, pp. 34-42, 1994.

[25] Terence Parr, *ANother Tool for Language Recognition (ANTLR)*, available at http://www.ANTLR.org.

[26] Ravi Rajwar, Alain Kagi, James Goodman. Private correspondence. The SimpleMP simulator was produced by the Galileo group at the University of Wisconsin - Madison.

[27] William J. Schmidt, Kelvin D. Nilsen, "Performance of a Hardware-Assisted Real-Time Garbage Collector," 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 76-85, Oct. 1994.

[28] James E. Smith, Timothy Heil, Subramanya Sastry, Todd M. Bezenek, "Achieving High Performance via Co-Designed Virtual Machines," Intl. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems , pp. 77-84, Oct. 1998.

[29] Y. Song, M. Dubois, "Assisted Execution," Technical Report #CENG 98-25, Dept. of EE-Systems, USC, Oct. 1998.

[30] "MAJC Architecture Tutorial", Sun Microsystems White Paper, May 1999.

[31] J. E. Thornton, "Parallel Operation in the Control Data 6600," American Federation of Information Processing Societies Conf. Proceedings, 26: Part II, FJCC, pp. 33-41, 1964.

[32] James O'Toole, Scott Nettles, "Concurrent Replicating Garbage Collection," 1994 ACM conference on LISP and Functional Programming, pp. 34-42, 1994.

[33] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", 23rd Intl. Symp. on Computer Architecture, pp. 191-202, May 1996.

[34] David Ungar, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," SIG-SOFT/SIGPLAN Practical Programming Environments Conf., pp. 157-167, April 1984.

[35] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," 1992 SIGPLAN Intl. Workshop on Memory Management, pp. 1-42, Sept. 1992.

[36] Mario Wolczko, Ifor Williams, "Multi-level Garbage Collection in a High-Performance Persistent Object System," 5th Intl. Workshop on Persistent Object Systems, Sept. 1992.

[37] W. Yamamoto, M.J. Serrano, A.R. Talcott, R.C. Wood, and M. Nemirosky, "Performance Estimation of Multistreamed, Superscalar Processors," 27th Hawaii Intl. Conf. on System Sciences, pp. I:105-204, Jan. 1994.

[38] W. Yamamoto, M. Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," Conf. on Parallel Architectures and Compilation Techniques, pp. 49-58, June 1995.

[39] Benjamin Zorn, "Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection," Proc. of the 1990 ACM Conf. on LISP and Functional Programming, pp. 87-98, June, 1990.

[40] Benjamin Zorn, "The Measured Cost of Conservative Garbage Collection," Univ. of Colorado, Boulder, CS Dept. Tech. Rep. CU-CS-573-92, Feb. 1992.