DOI:10.1145/2534706.2534725

Technical Perspective
The Cleanest Garbage Collection

By Eliot Moss

GARBAGE COLLECTION, THE quirky name used for automatic storage management, might well be called memory recycling if it were invented today. Indeed, it has a venerable history, nearly as long as that of computing itself, extending back to early LISP implementations, with papers appearing from 1960 onward. GC, as it is affectionately known, also developed a reputation for being slow and requiring a lot more memory than explicitly managed memory. If that was not enough, most GC algorithms would wait for the designated memory to fill, then stop the program, collect, and restart the program, introducing pauses into the primary computation.

Proponents of GC have persuasive software engineering arguments on their side: automatic storage reclamation simplifies programming and reduces errors, thus increasing programmer productivity. Ultimately, increasing computer speed and memory capacity made GC a reasonable choice for a much wider range of systems. Java brought it into the mainstream, where it has been quite successful. We have even reached the point where C++, originally posted with signs reading "No GC here!", now offers optional support for it.

But the holy grail for automatic storage management has been to achieve GC for *real-time* systems. Real-time GC is challenging for several reasons. One is that real-time systems cannot tolerate large pauses. This requires the collector either to be incremental at a fine grain or to run concurrently with the program (called the *mutator* in GC parlance because it nefariously changes pointers around while the GC is at work). I can recall when I was a graduate student it seemed there was a veritable industry around presenting concurrent GC algorithms in this very publication, with proofs of correctness. In fact, these proofs were offered because they were among the hardest correctness proofs researchers could conceive. Needless to say, this suggests the difficulty of getting concurrent GC algorithms right, much less translating them into correct implementations.

It is quite a tour de force that the authors of the following paper have built a provably correct real-time collector for reconfigurable hardware.

Concurrent GC alone is not enough to achieve real-time storage management. You also need provable bounds on the time to collect and the maximum memory needed by your program running under this scheme. Firstly, the collector cannot fall behind the mutator: it must be able to collect disused memory and recycle it for mutator use at least as fast as the mutator allocates memory units. An implication is that unless you can impose an artificial bound on the mutator's allocation rate, your collector must not only be concurrent but also *fast*.

While people have developed a wide range of GC algorithms, we are concerned here with ones that start from program variables (*roots*) and follow pointers from object to object, finding all the *reachable* objects. Such *tracing collectors* work in cycles: from the roots, trace the reachable objects, reclaim what is left, then go on to the next cycle.

Though it is a bit of a misnomer, GC developers also call reachable objects *live*, and their total volume at a given point in execution is the *live size*. In addition to a fast enough concurrent GC algorithm, for real-time GC you need not only a hard bound on the program's maximum live size—probably needed for real-time mutator behavior anyway—but also a bound on the amount of garbage (*unreachable* objects) that will accumulate during a collection cycle.

It is thus quite a tour de force that the authors of the following paper have built a provably correct real-time collector for reconfigurable hardware (field programmable gate arrays). How can this be? It turns out the FPGA setting makes the problem simpler in some ways than is the case with software GC running on stock processors. They can reasonably impose simple uniformity on the memory and its layout; they can exploit single-clock read-modify-write steps; and perhaps most importantly they have, via dual ported memory, completely concurrent memory access. This all leads to one of the cleanest and perhaps most understandable implementations of a concurrent GC that has ever been presented. The other prerequisites for real-time collection also follow easily. It is difficult to find the right word to express the feeling I get seeing such a sophisticated algorithmic idea reduced to such a straightforward hardware implementation—"Cool!" will have to suffice.

This work does not appear to offer a direct path to simple real-time GC support for software implementations on stock hardware. At the same time, it helps to inform that work through its contribution to knowledge about real-time GC beyond its benefits to practice. In particular, it shows how tight a bound we can achieve on the total space needed for a no-pause collector to run. I feel certain it will inspire creative approaches that will help bring garbage collection into acceptance in almost every corner of the system implementation space.

Eliot Moss (moss@cs.umass.edu) is a professor in the Department of Computer Science at the University of Massachusetts, Amherst.

Copyright held by Author/Owner(s).

DOI:10.1145/2534706.2534726

And Then There Were None: A Stall-Free Real-Time Garbage Collector for Reconfigurable Hardware

By David F. Bacon, Perry Cheng, and Sunil Shukla

1. INTRODUCTION

The end of frequency scaling has driven architects and developers to parallelism in search of performance. However, general-purpose MIMD parallelism can be inefficient and power-hungry, with power rapidly becoming the limiting factor. This has led the search for performance to non-traditional chip architectures like GPUs and other more radical architectures. The most radical computing platform of all is reconfigurable hardware, in the form of Field-Programmable Gate Arrays (FPGAs).

FPGAs are now available with over one million programmable logic cells and 8MB of on-chip "Block RAM," providing a massive amount of bit-level parallelism combined with single-cycle access to on-chip memory. Furthermore, because that memory is distributed over the chip in distinct 36Kb units, the potential internal bandwidth is very high.

However, programming methodology for FPGAs has lagged behind their capacity, consequently reducing their applicability to general-purpose computing. The common languages in this domain are still hardware description languages (VHDL and Verilog) in which the only abstractions are bits, arrays of bits, registers, wires, and so on. The entire approach to programming them is oriented around the synthesis of a chip that happens to be reconfigurable, as opposed to programming a general-purpose device.

Recent research has focused on raising the level of abstraction and programmability to that of high-level software-based programming languages: the Scala-based Chisel framework,[4] C#-based Kiwi project[10] and the Liquid Metal project, which has developed the Lime language[3] based on Java. However, until now, whether programmers are writing in low-level HDLs or high-level languages like Chisel and Lime, use of dynamic memory management has only just begun to be explored,[8, 14] and use of garbage collection has been nonexistent.

In this article, we present a garbage collector synthesized directly to hardware, capable of collecting a heap of *uniform* objects completely concurrently. These uniform heaps (*miniheaps*) are composed entirely of objects of a fixed shape. Thus, the size of the data fields and the location of pointers of each object are fixed. We trade some flexibility in the memory layout for large gains in collector performance. In the FPGA domain, this trade-off makes sense: due to the distributed nature of the memory, it is common to build pipelined designs where each stage of the pipeline maintains its own internal data structures that are able to access their local block RAM in parallel with other pipeline stages. Furthermore, fixed data layouts can provide order-of-magnitude better performance because they allow designs which deterministically process one operation per clock cycle.

Algorithmically, our collector is a Yuasa-style snapshot-at-the-beginning collector,[16] with a linear sweep. By taking advantage of hardware structures like dual-ported memories, the ability to simultaneously read and write a register in a single cycle, and to atomically distribute a control signal across the entire system, we are able to develop a collector that *never* interferes with the mutator. Furthermore, the mutator has single-cycle access to memory. Arbitration circuits delay some collector operations by one cycle in favor of mutator operations, but the collector can keep up with a mutator even when it performs a memory operation every cycle (allocations are limited to one every other cycle).

Our stall-free collector can be used directly with programs hand-written in hardware description languages. Alternatively, it can be part of a hardware "runtime system" used by high-level language[3, 10] systems including dynamic memory allocation. For evaluation purposes, we have also implemented a stop-the-world variant and a malloc/free-style system. Using a very allocation-intensive application, the three collectors are compared in terms of memory usage, clock frequency, throughput (cycles and wall-clock time), and application stalls. We also present analytic closed-form worst-case bounds for the minimum heap size required for 0-stall real-time behavior, which are empirically validated. Further details, such as the energy consumption and additional benchmarking, are available in the original paper.[6]

2. FPGA BACKGROUND

FPGAs are programmable logic devices consisting of many discrete resources units that are enabled and connected based on the user application. Resources include look-up tables (LUTs) which can be used to implement combinational logic, and flip-flops which can be used to implement sequential logic or

The original version of this paper was published in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, June 2012, ACM.

state. On the Xilinx FPGAs, which we use in this work, LUTs and flip-flops are grouped into *slices*, which is the standard unit in which resource consumption is reported for FPGAs.

Particularly important to this work are the discrete memory blocks, called Block RAMs (BRAMs), available on the FPGA. BRAMs are dual-ported specialized memory structures embedded within the FPGA for resource-efficient implementation of random- and sequential-access memories.

The Xilinx Virtex-5 LX330T[15] device that we use in this paper has a total BRAM capacity of 1.45MB. A single BRAM in a Virtex-5 FPGA can store up to 36Kb of memory. An important feature of BRAM is that it can be organized in various form factors (addressable range and data width combinations). For example, a 36Kb BRAM can be configured as 36K × 1 (36K 1-bit locations), 16K × 2, 8K × 4, 4K × 9, 2K × 18, or 1K × 36 memory.

A 36Kb BRAM can also be used as two independent 18Kb BRAMs. Larger logical memory structures can be built by cascading multiple BRAMs. Any logical memory structure in the design which is not a multiple of 18Kb would lead to quantization (or, in memory system parlance, "fragmentation").

The quantization effect can be considerable depending on the logical memory structure in the design (and is explored in Section 7). A BRAM can be used as a true dual ported (TDP) RAM providing two fully independent read-write ports. Furthermore, each port supports the following three modes for the read-write operation: read-before-write (previously stored data is available on the read bus), read-after-write (newly written data is available on the read bus), and no-change (read data bus output remains unchanged). Our collector makes significant use of read-before-write for features like the Yuasa-style write barrier.[16]

BRAMs can also be configured for use as FIFOs rather than as random access memories; we make use of this feature for implementing the mark queues in the tracing phase of the collector.

FPGAs are typically packaged on boards with dedicated off-chip DRAM and/or SRAM which can be accessed via a memory controller synthesized for the FPGA. Such memory could be used to implement much larger heap structures. However, we do not consider use of DRAM or SRAM in this paper because we are focusing on high-performance designs with highly deterministic (single cycle) behavior.

3. MEMORY ARCHITECTURE

The memory architecture—that is, the way in which object fields are laid out in memory, and the free list is maintained—is common to our support of both malloc/free and garbage-collected abstractions. In this section we describe our memory architecture as well as some of the alternatives, and discuss the trade-offs qualitatively. Some trade-offs are explored quantitatively in Section 7.

Since memory structures within an FPGA are typically and of necessity far more uniform than in a conventional software heap, we organize memory into one or more *miniheaps*, in which objects have a fixed size and "shape" in terms of division between pointer and data fields.

3.1. Miniheap interface

Each miniheap has an interface allowing objects allocation (and freeing when using explicit memory management), and operations for individual data fields to be read or written. In this article, we consider only miniheaps with one or two pointer fields and one or two data fields. This is sufficient for implementing stacks, lists, queues, and tree data structures. FPGA modules for common applications like packet processing, compression, etc. are covered by such structures.

Our design allows an arbitrary number of data fields. Increasing the number of pointer fields is straightforward for malloc-style memory. However, for garbage collected memory, the extension would require additional logic. We believe this is relatively straightforward to implement (and include details below) but the experimental results in this paper are confined to one- and two-pointer objects.

3.2. Miniheap with malloc/free

There are many ways in which the interface in Section 3.1 can be implemented. Fundamentally, these represent a time/space trade-off between the number of available parallel operations, and the amount of hardware resources consumed.

For FPGAs, one specifies a logical memory block with a desired data width and number of entries, and the synthesis tools attempt to allocate the required number of individual Block RAMs as efficiently as possible, using various packing strategies. We refer to the BRAMs for such a logical memory block as a *BRAM set*.

In our design we use one BRAM set for each field in the object. For example, if there are two pointer fields and one data field, then there are three BRAM sets.

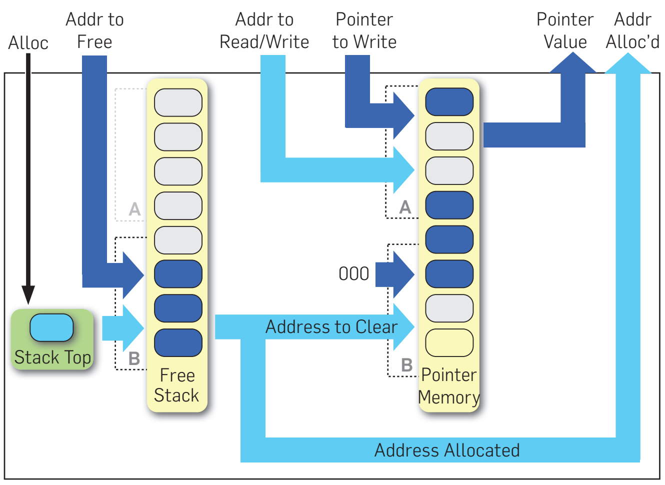
The non-pointer field has a natural width associated with its data type (for instance 32 bits). However, for a miniheap of size N, the pointer fields need only be $\lceil \log_2 N \rceil$ bits wide. Because data widths on the FPGA are completely customizable, we use precisely the required number of bits. Thus a larger miniheap will increase in size not only because of the number of entries, but because the pointer fields themselves become larger. As in software, the pointer value 0 is reserved to mean "`null`", so a miniheap of size N can really only store $N - 1$ objects.

A high-level block diagram of the memory manager is shown in Figure 1. It shows the primary data and control fields of the memory module, although many of the signals have been elided to simplify the diagram. For clarity of presentation it shows a single object field, of pointer type (Pointer Memory), which is stored in a single BRAM set. A second BRAM set (Free Stack) is used to store a stack of free objects. Using the read-before-write property of BRAMs we are able to implement the two functions (alloc and free) using port B of the BRAM set (Free Stack), leaving port A unused.

For an object with f fields, there are f BRAM sets with associated interfaces for the write and read values (but *not* an additional address port). There is only a single free stack, regardless of how many fields the object has.

The *Alloc* signal is a one-bit signal used to implement the `malloc` operation. A register is used to hold the value of the stack top. Assuming it is non-zero, it is decremented and then presented on port B of the Free Stack BRAM set, in *read* mode. The resulting pointer to a free field is then returned (*Addr Alloc'd*), but is also fed to port B of the Pointer Memory, in *write* mode with the write value

Figure 1. Memory module design for malloc/free interface, showing a single pointer field. BRAMs are in yellow, with the dual ports (A/B) shown. Ovals designate pointer fields; those in blue are in use.



hard-wired to 000 (or "`null`").

To free an object, the pointer is presented to the memory manager (*Addr to Free*). The Stack Top register is used as the address for the Free Stack BRAM set on port B, in write mode, with the data value *Addr to Free*. Then the Stack Top register is incremented. This causes the pointer to the freed object to be pushed onto the Free Stack.

In order to read or write a field in the Pointer Memory, the *Addr to Read/Write* is presented, and, if writing, a *Pointer to Write*. This uses port A of the BRAM set in either read or write mode, returning a value on the *Pointer Value* port in the former case.

Note that this design, by taking advantage of the dual-porting of the BRAMs, allows a read or write to proceed in parallel with an allocate or free.

Threaded free list. A common software optimization would be to represent the free objects not as a stack of pointers, but as a linked list threaded through the unused objects (that is, a linked list through the first pointer field). Since the set of allocated and free objects are mutually exclusive, this space optimization is essentially free modulo cache locality effects.

However, in hardware, this causes resource contention on the BRAM set containing the first pointer (since it is doing double duty). Thus parallelism is reduced: read or write operations on the first pointer cannot be performed in the same cycle as `malloc` or `free`, and the latter require two cycles rather than one. Our choice in using a stack is deliberate as the modest increased use of space is far less costly than the potential resource contention.

4. GARBAGE COLLECTOR DESIGN

We now describe the implementation of both a stop-the-world and a fully concurrent collector in hardware. In software, the architectures of these two styles of collector are quite different. In hardware, the gap is much smaller, as the same fundamental structures and interfaces are used.

The concurrent collector has a few extra data structures (implemented with BRAMs) and requires more careful allocation of BRAM ports to avoid contention, but these features do not negatively affect its use by the stop-the-world collector. Therefore, we will present the concurrent collector design, and merely mention here that the stop-the-world variant omits the shadow register(s) from the root engine, the write barrier register and logic from the trace engine, and the used map and logic from the sweep engine. Our design comprises three separate components, which handle the atomic root snapshot, tracing, and sweeping.

4.1. Background: Yuasa's snapshot algorithm

Before delving into the details of our implementation, we describe Yuasa's snapshot algorithm[16] which is the basis of our implementation. While the mechanics in hardware are quite different, it is interesting to note that implementing in hardware allows us to achieve a higher degree of concurrency and determinism than state-of-the-art software algorithms, but without having to incorporate more sophisticated algorithmic techniques developed in the interim.

The fundamental principle of the snapshot algorithm is that when collection is initiated, a *logical* snapshot of the heap is taken. The collector then runs in this logical snapshot, and collects everything that was garbage at snapshot time.

In Yuasa's original algorithm, the snapshot consisted of the registers, stacks, and global variables. This set of pointers was gathered synchronously (since then, much research has been devoted to avoiding the need for any global synchronization at snapshot time or during phase transitions[2]).

Once the roots have been gathered, the mutator is allowed to proceed and the collector runs concurrently, marking the transitive closure of the roots.

If the mutator concurrently modifies the heap, its only obligation is to make sure that the collector can still find all of the objects that existed in the heap at snapshot time. This is accomplished by the use of a *write barrier*: before any pointer is overwritten, it is recorded in a buffer and treated as a root for the purposes of collection.

Objects that are freshly allocated during a collection are not eligible for collection (they are "allocated black" in the parlance of collector literature).

The advantages of the snapshot algorithm are simplicity and determinism. Since it operates on a logical snapshot at an instant in time, the invariants of the algorithm are easy to describe. In addition, termination is simple and deterministic, since the amount of work is bounded at the instant that collection begins.

4.2. Root snapshot

The concurrent collector uses the snapshot-at-the-beginning algorithm described above. Yuasa's original algorithm required a global pause while the snapshot was taken by recording the roots; since then real-time collectors have endeavored to reduce the pause required by the root snapshot. In hardware, we are able to completely eliminate the snapshot pause by taking advantage of the parallelism and synchronization available in the hardware.

The snapshot must take two types of roots into account: those in registers, and those on the stack. Figure 2 shows the root snapshot module, simplified to show a single stack and a single register.

The snapshot is controlled by the *GC* input signal, which goes high for one clock cycle at the beginning of collection. The snapshot is defined as the state of the memory at the beginning of the *next cycle* after the *GC* signal goes high. This allows some setup time and reduces synchronization requirements. Note that if the *GC* signal is asserted while a collection is already under way, then the trigger is ignored.

The register snapshot is obtained by using a shadow register. In the cycle after the *GC* signal goes high, the value of the register is copied into the shadow register. This can happen even if the register is also written by the mutator in the same cycle, since the new value will not be latched until the end of the cycle.

The stack snapshot is obtained by having another register in addition to the Stack Top register, called the Scan Pointer. In the same cycle that the *GC* signal goes high, the value of the Stack Top pointer minus one is written into the Scan Pointer (because the Stack Top points to the entry above the actual top value). Beginning in the following cycle, the Scan Pointer is used as the source address to port B of the BRAM set containing the stack, and the pointer is read out, going through the MUX and emerging on the *Root to Add* port from the snapshot module. The Scan Pointer is also decremented in preparation for the following cycle.

Note that the mutator can continue to use the stack via port A of the BRAM set, while the snapshot uses port B. And since the mutator cannot pop values off the stack faster than the collector can read them out, the property is preserved that the snapshot contains *exactly* those roots that existed in the cycle following the *GC* signal.

A detail omitted from the diagram is that a state machine is required to sequence the values from the stack and the shadow register(s) through the MUX to the *Root to Add* port. Note that the values from the stack must be processed first, because the stack snapshot technique relies on staying ahead of the mutator without any explicit synchronization.

If multiple stacks were desired, then a "shadow" stack would be required to hold values as they were read out before the mutator could overwrite them, which could then be sequenced onto the *Root to Add* port.

As will be seen in Section 4.4, collection is triggered (only) by an allocation that causes free space to drop below a threshold. Therefore the generation of root snapshot logic only needs to consider those hardware states in which this might occur. Any register or stack not live in those states can be safely ignored.

4.3. Tracing

The tracing engine, along with a single pointer memory (corresponding to a single pointer field in an object) is shown in Figure 3. It provides the same mutator interface as the malloc/free style memory manager of Figure 1: *Addr to Read/Write*, *Pointer to Write*, and *Pointer Value*—except that the external interface *Addr to Free* is replaced by the internal interface (denoted in red) *Addr to Clear*, which is generated by the Sweep module (described in Section 4.4).

The only additional interface is the *Root to Add* port which takes its inputs from the output port of the same name of the Root Engine in Figure 2.

As it executes, there are three sources of pointers for the engine to trace: externally added roots from the snapshot, internally traced roots from the pointer memory, and overwritten pointers from the pointer memory (captured with a Yuasa-style barrier to maintain the snapshot property). The different pointer sources flow through a MUX, and on each cycle a pointer can be presented to the Mark Map, which contains one bit for each of the *N* memory locations.

Using the BRAM read-before-write mode, the old mark value is read, and then the mark value is unconditionally set to 1. If the old mark value is 0, this pointer has not yet been traversed, so the negation of the old mark value (indicated by the bubble) is used to control whether the pointer is added to the Mark Queue (note that this means that all values in the Mark Queue have been filtered, so at most $N - 1$ values can flow through the queue).

Pointers from the Mark Queue are presented as a read address on port B of the Pointer Memory, and if the fetched

Figure 2. Atomic root snapshot engine.

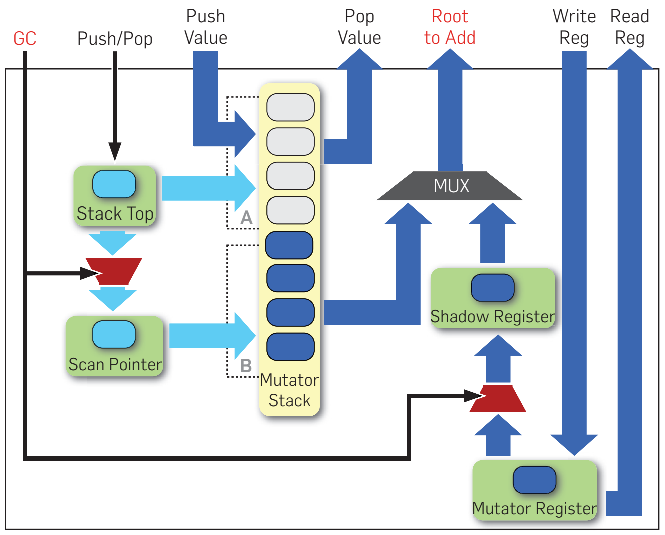
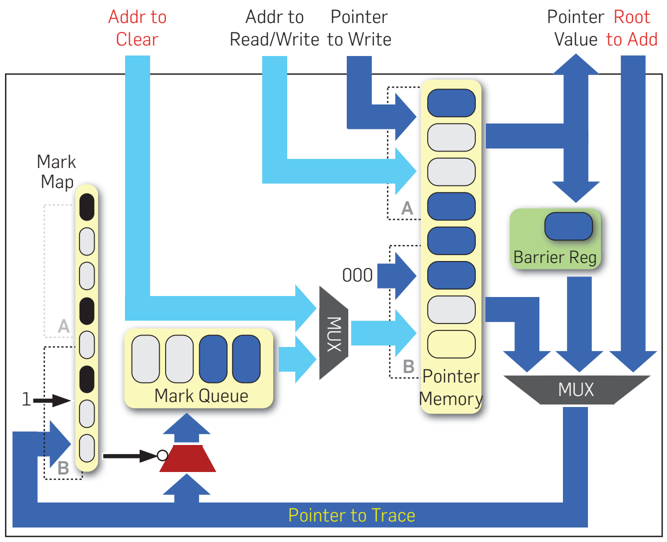


Figure 3. Tracing engine and one-pointer memory.



values are non-null, they are fed through the MUX and thence to the marking step.

The write barrier is implemented by using port A of the Pointer Memory BRAM in read-before-write mode. When the mutator writes a pointer, the old value is read out first and placed into the Barrier Reg. This is subsequently fed through the MUX and marked (the timing and arbitration is discussed below).

Given the three BRAMs involved in the marking process, processing one pointer requires 3 cycles. However, the marking engine is implemented as a *3-stage pipeline*, so it is able to sustain a throughput of one pointer per cycle.

Trace engine pairing. For objects with two pointers, two trace engines are paired together to maximize resource usage (this is not shown in the figure). Since each trace engine only uses one port of the mark map, both engines can mark concurrently.

The next item to mark is always taken from the longer queue. When there is only one item to enqueue, it is placed on the shorter queue. Using this design, we provision each of the 2 queues to be of size $3\,N/8 + R$ (where R is the maximum number of roots), which guarantees that the queues will never overflow.

On each cycle, one pointer is removed from the queues, and the two pointers in the object retrieved are examined and potentially marked and enqueued.

To minimize interference due to write barrier, our design has two write barrier registers. The write barrier values are not processed until a pair is formed and coupled with the fact that there are two mark queues, the mark engines can make progress every other cycle even if the application is performing one write per cycle.

Trace termination and WCET effects. The termination protocol for marking is simple: once the last item from the mark queues is popped (both mark queues become empty), it takes 2 or 3 cycles for the trace engine to finish the current pipeline. If the two pointers returned by the heap are null, then the mark process is terminated in the second cycle as there is no need to read the mark bits in this case. Otherwise the mark bit for the non-null pointers are read to ensure that both pointers are marked, in which case the mark phase is terminated in the third cycle.

Write barrier values arriving after the first cycle of termination can be ignored, since by the snapshot property they would either have to be newly allocated or else discovered by tracing the heap.

However, note that some (realistic) data structures, in particular linked lists, will cause a pathological behavior, in which a pointer is marked, removed from the queue, which will appear empty, and then 2 cycles later, the next pointer from the linked list will be enqueued. So while the pipeline can sustain marking one object per cycle, pipeline bubbles will occur which reduce that throughput.

4.4. Sweeping

Once tracing is complete, the sweep phase begins, in which memory is reclaimed. The high-level design is shown in Figure 4. The sweep engine also handles allocation requests and maintains the stack of pointers to free memory (Free Stack). The Mark Map here is the same Mark Map as in Figure 3.

When an *Alloc* request arrives from the mutator, the Stack Top register is used to remove a pointer to a free object from the Free Stack, and the stack pointer is decremented. If the stack pointer falls below a certain level (we typically use 25%), then a garbage collection is triggered by raising the *GC* signal which is connected to the root snapshot engine (Figure 2).

The address popped from the Free Stack is returned to the mutator on the *Addr Alloc'd* port. It is also used to set the object's entry in the Used Map, to `01`, meaning "freshly allocated" (and thus "black"). A value of `00` means "free", in which case the object is on the Free Stack.

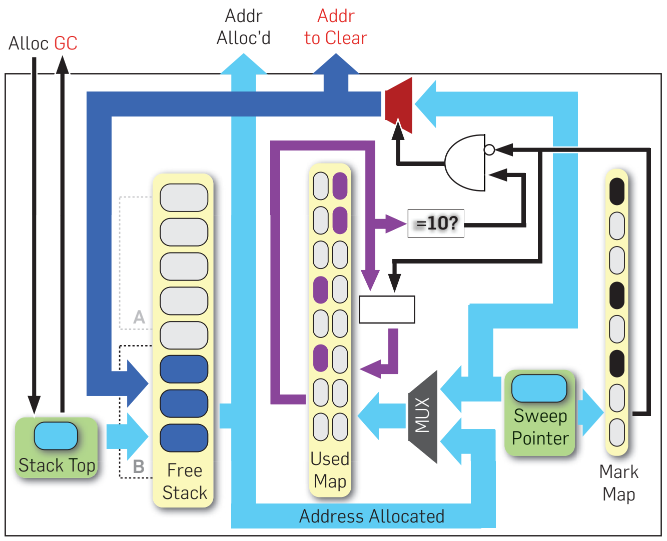
When tracing is completed, sweeping begins in the next cycle. Sweeping is a simple linear scan. The Sweep Pointer is initialized to 1 (since slot 0 is `null`), and on every cycle (except when preempted by allocation) the sweep pointer is presented to both the Mark Map and the Used Map.

If an object is marked, its Used Map entry is set to `10`. If an object is not marked and its used map entry is `10` (the *and* gate in the figure) then the used map entry is reset to `00`. Although only 3 states are used, the particular choice of bit pattern avoids unneeded logic in the critical path. The resulting signal is also used to control whether the current Sweep Pointer address is going to be freed. If so, it is pushed onto the Free Stack and also output on the *Addr to Clear* port, which is connected to the mark engine so that the data values being freed are zeroed out.

Note that since clearing only occurs during sweeping, there is no contention for the Pointer Memory port in the trace engine between clearing and marking. Furthermore, an allocation and a free may happen in the same cycle: the top-of-stack is accessed using read-before-write mode and returned as the *Addr Alloc'd*, and then the newly freed object is pushed back.

When an object is allocated, its entry in the Mark Map is *not* set (otherwise an extra interlock would be required). This means that the tracing engine may encounter newly allocated objects in its marking pipeline (via newly installed pointers in the heap), albeit at most once since they will then be marked. This also affects WCET analysis, as we will see in the next section.

Figure 4. Free stack and sweeping engine.



5. REAL-TIME BEHAVIOR

First of all, we note that since the design of our real-time collector allows mutation and collection to occur unconditionally together in a single cycle, the minimum mutator utilization (or MMU[7]), is 100% unless insufficient resources are dedicated to the heap.

Furthermore, unlike software-based collectors,[5, 11] the system is fully deterministic because we can analyze the worst case behavior down to the (machine) cycle.

Given R is the maximum number of roots, N is the size of the heap, then the worst-case time (in cycles) for garbage collection is

$$T = T_R + T_M + T_W + T_X + T_S + T_A \qquad (1)$$

where T_R is the time to snapshot the roots, T_M is the time (in cycles) to mark, T_S is the time to sweep, and T_W is the time lost to write barriers during marking, T_X is the time lost to blackening newly allocated objects during marking, and T_A is time lost to allocations during sweeping.

In the worst case, without any knowledge of the application,

$$T_R = R+2 \quad T_M = 3N + 3 \quad T_W = 0 \quad T_X = 0 \quad T_S = N$$

The reasoning for these quantities follows. During the snapshot phase, we can place one root into the mark queue every cycle, plus one cycle to start and finish the phase, accounting for $R + 2$. During marking, there could be N objects in the heap, configured as a linked list which caused the mark pipeline to stall for two cycles on each object, plus 3 cycles to terminate. Sweeping is unaffected by application characteristics, and always takes N cycles. Preemption of the collector by mutator write barriers (T_W) does not factor into the worst-case analysis because the write barrier work is overlapped with the collector stalls. Extra mark operations to blacken newly allocated objects (T_X) also simply fill stall cycles.

Our design allows an allocation operation *in every cycle*, but allocation preempts the sweep phase, meaning that such an allocation rate can only be sustained in short bursts. The largest sustainable allocation rate is 0.5—otherwise the heap would be exhausted before sweeping completed. Thus $T_A = N$ and

$$T_{\text{worst}} = R + 5N + 5 \qquad (2)$$

5.1. Application-specific analysis

Real-time analysis typically takes advantage of at least some application-specific knowledge. This is likely to be particularly true of hardware-based systems. Fortunately, the structure of such systems makes it more likely that such factors can be quantified to a high degree of precision, for example by looking at operations per clock cycle in the synthesized design.

Let μ be the average number of mutations per cycle ($\mu \leq 1$), α be the average number of allocations per cycle ($\alpha < 0.5$), and m be the maximum number of live data objects in the heap at any one time ($m < N$). Then we can more precisely estimate

$$T'_M = 3m+3 \quad T'_X = \alpha T'_M \quad T'_W = \frac{\mu}{2-\mu} m \quad T'_A = \frac{\alpha}{1-\alpha} N$$

Note that both α and μ can only be averaged over a *time window* guaranteed to be less than or equal to the phases which they influence; m is a safe window size.

The largest inaccuracy is still due to pipeline stalls during marking, for which worst- and average-case behavior can be very different. We therefore let B be the number of pipeline stalls ($0 \leq B \leq 2m$), so an even more precise bound on marking is $T''_M = m+B+3$ (and also improving $T''_X = \alpha\, T''_M$).

For a linked list, $B = 2m$; for three linked lists each with its own root, $B = 0$. We hypothesize that for the heap considered as a forest without back-edges, B is bounded by the number of levels of width 1 plus the number of levels of width 2 (when the width is 3 or greater, there is enough parallelism to keep the 3-stage pipeline full and avoid stalls).

Using these application-specific estimates, we then are able to bound the worst-case execution time (WCET) of collection as

$$T_{\max} = \left(\frac{1}{1-\alpha}\right)\left(R+B+5+\frac{2}{2-\mu}m+\frac{N}{1-\alpha}\right) \qquad (3)$$

5.2. Minimum heap size

Once the worst-case execution time for collection is known, we can solve for the minimum heap size in which the collector can run with real-time behavior (zero stalls). Note that if a heap size is deliberately chosen below this minimum, the allocator may experience an out-of-memory condition. As a starting point, m objects must be available for the live data. While a collection taking time $T_{\max}$ takes place, another $\alpha T_{\max}$ objects can be allocated. However, there may also be $\alpha T_{\max}$ floating garbage from the previous cycle when a collection starts. Thus the minimum heap size is

$$N_{\min} = m + 2\alpha\, T_{\max} \qquad (4)$$

and if we denote the non-size-dependent portion of $T_{\max}$ from equation (3) by

$$K = \left(\frac{1}{1-\alpha}\right)\left(R+B+5+\frac{2}{2-\mu}m\right)$$

then we can solve for

$$\begin{aligned} N_{\min} &= m+2\alpha T_{\max} \\ &= m+2\alpha\left(K+\frac{N_{\min}}{(1-\alpha)^2}\right) \\ N_{\min} &= \frac{(1-\alpha)^2(m+2\alpha K)}{1-4\alpha+\alpha^2} \end{aligned} \qquad (5)$$

6. EXPERIMENTAL METHODOLOGY

Since we have implemented the first collector of this kind, we cannot leverage a preexisting set of benchmarks for evaluation. Here, we present the results of a very allocation-intensive microbenchmark, a doubly-ended queue (deque), which commonly occurs in many applications. In our HDL implementation, the doubly-linked list can be modified by pushes and pops to either the front or back. The workload consists of a pseudorandom sequence of such operations while keeping the maximum amount of live data close to but no more than 8192. Because there is almost no computation performed on the list elements, this benchmark is very allocation-intensive. Moreover, the linear nature of the data structure prevents the collector from taking advantage of graph fanout. These two factors combine to present a great challenge to the collector.

It is important to note that because of the very high degree of determinism in the hardware, and in our collector implementation, such micro-benchmarks can provide a far more accurate picture of performance than in typical evaluations of CPU-based collectors running in software. There are no cache effects, no time-slicing, and no interrupts. Because these higher order effects are absent, the performance behavior presented to the mutator by the collector and vice versa is completely captured by the memory management API at a cycle-accurate level. We validate this experimentally by showing that the estimates for collection time and minimum real-time heap size (from Section 5.1) are highly accurate.

A given micro-benchmark can be paired with one of the three memory management implementations (Malloc, stop-the-world GC, and real-time GC). Furthermore, these are parameterized by the size of the miniheap, and for the collectors, the trigger at which to start collection (although for most purposes, we simply trigger when free space falls below 25%). We call these *design points*.

Our experiments are performed by using the Xilinx ISE 13.4 synthesis tools on a Xilinx Virtex-5 LX330T,[15] which is the largest FPGA within the Virtex5 LXT product line. The LX330T has 51,840 slices and 11,664Kb (1.45MB) of Block RAM. Fabricated in 65 nm technology, the chip is theoretically capable of being clocked at up to 550 MHz, but realistic designs generally run between 100 and 300 MHz.

7. EVALUATION

We begin by examining the cost, in terms of static resources, of the 3 memory managers—malloc/free ("Malloc"), stop-the-world collection ("STW"), and real-time concurrent collection ("RTGC"). For these purposes we synthesize the memory manager in the absence of any application. This provides insight into the cost of the memory management itself, and also provides an upper bound on the performance of actual applications (since they can only use more resources or cause the clock frequency to decline).

We evaluate design points at heap sizes (in objects) from 1K to 64K in powers of 2. For these purposes we use an object layout of two pointers and one 32-bit data field. For brevity, we omit detailed usage of non-BRAM logic resources. It is enough to note that for all cases, the logic consumption is under 1% for all 3 variants and at all heaps sizes.

Figure 5 shows BRAM consumption. Because we have chosen powers of 2 for heap sizes, the largest heap size only uses 60% of the BRAM resources (one is of course free to choose other sizes). At the smaller heap sizes, garbage collectors consume up to 80% more BRAMs than Malloc. However, at realistic heap sizes, the figure drops to 24%. In addition, RTGC requires about 2–12% more memory than STW since it requires the additional 2-bit wide Used Map to cope with concurrent allocation. Fragmentation is noticeable but not a major factor, ranging from 11–31% for Malloc and 11–53% for garbage collection. As before, at larger heap sizes, the fragmentation decreases. Some wastage can be avoided by choosing heap sizes more carefully, not necessarily a power of 2, by noting that BRAMs are available in 18Kb blocks. However, some fragmentation loss is inherent in the quantization of BRAMs as they are chained together to form larger memories.

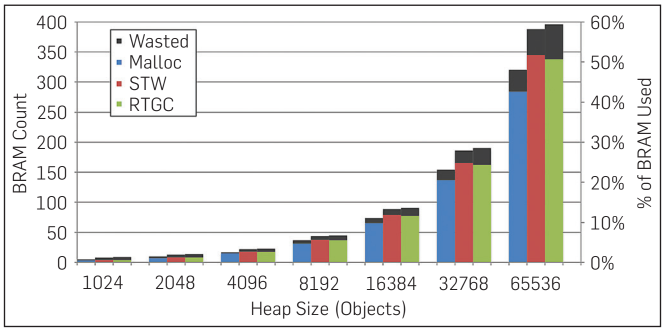
Finally, Figure 6 shows the synthesized clock frequency at different design points. Here we see a significant effect from the more complex logic for garbage collection: even though it consumes relatively little area, clock frequency for garbage collection is noticeably slower (15–39%) than Malloc across all design points. On the other hand, the difference between STW and RTGC is small with RTGC often faster. Regardless of the form of memory management, clock frequency declines as the heap becomes larger. However, the overall clock rate may very well be constrained by the application logic rather than the collector logic, as we will see below.

7.1. Throughput

So far we have discussed the costs of memory management in the absence of applications; we now consider what happens when the memory manager is "linked" to the Deque microbenchmark. Unlike the previous section, where we concentrated on the effects of a wide range of memory sizes on static chip resources, here we focus on a smaller range of sizes using a trace with a single maximum live data set of $m = 8192$ as described previously. We then vary the heap size N from m to $2m$ at fractional increments of $m/10$. As we make memory scarce, the resolution is also increased to $m/100$ to show how the system behaves at very tight conditions. Each design point requires a full synthesis of the hardware design which can affect the frequency, power, and execution time.

Figure 7 shows the throughput of the benchmark as the heap size varies for all 3 schemes. To understand the interaction of various effects, we not only examine the throughput both in cycle duration but also, since the synthesizable clock frequencies vary, in physical time. The Deque benchmark shows a different behavior. With much higher allocation and mutation rates ($\alpha = 0.07$ and $\mu = 0.13$), it is much more sensitive to collector activity. As seen in Figure 7(b), even at heap size $N = 2m$, STW consumes noticeably more cycles, rising to almost double the cycles

Figure 5. Block RAM usage, including Fragmentation.



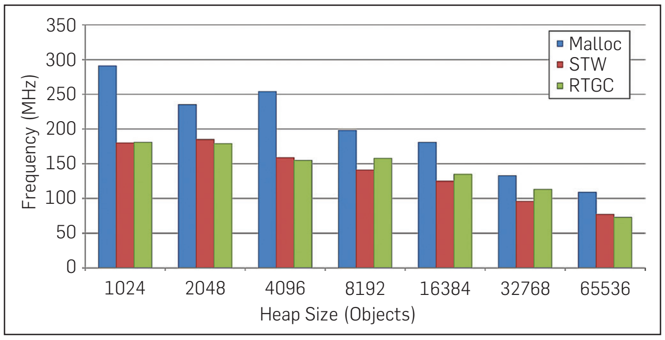
at $N = 1.1m$. By contrast RTGC consumes slightly fewer cycles than Malloc until it begins to experience stall cycles (non-real-time behavior) at $N = 1.4m$ because it falls behind the mutator.

The Deque benchmark has a very simple logic so any limitations on frequency introduced by the collector is magnified. The effect is seen clearly in Figure 7(b): Malloc synthesizes at a higher frequency, allowing it to make up RTGC's slight advantage in cycles and consume 25% less time on an average. STW suffers even more from the combined effect of a lower clock frequency and additional cycles due to synchronous collection. On average, RTGC is faster than STW by 14% and never interrupts the application.

These measurements reveal some surprising trends that are completely contrary to the expected trade-offs for software collectors: RTGC is actually *faster, more deterministic, and requires less heap space* than STW! There seems to be no reason to use STW because the natural advantage of implementing concurrency in hardware completely supersedes the traditional latency versus bandwidth trade-off.

Furthermore, RTGC allows applications to run at far lower multiples of the maximum live set m than possible for either real-time or stop-the-world collectors in software. RTGC is also only moderately slower than Malloc, meaning that the cost of abstraction is quite palatable. We do not show the results for other applications but note that, as predicted, this performance gap decreases as the application becomes more complex.

Figure 6. Synthesized clock frequency.



7.2. Validation of real-time bounds

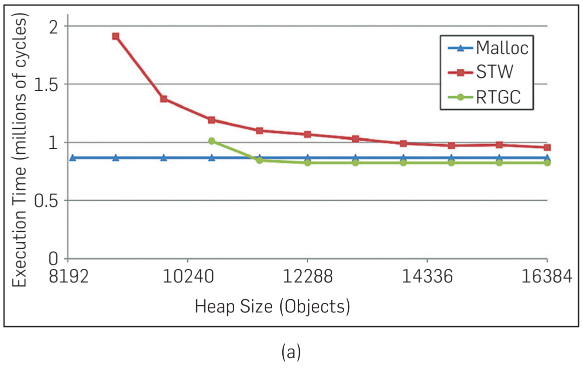
Because the design and analysis of our concurrent collector is intended to be cycle-accurate, we can validate the time and space bounds of the collector with expectation that they will be fully met. Figure 8 shows the actual time spent in garbage collection and the analytic upper bound (T_{max} from equation 3). Note that we show both average as well as the maximum time spent in garbage collections. The heap size is chosen to be much tighter than in earlier graphs as our focus here is how the collector behaves when it is under stress (near N_{min} from equation 4). For convenience, we express heap size as a fraction (N/m) of the maximum amount of live data since our bounds are almost linear when considered in terms of m and N.

Average time spent in collection is always less than the predicted worst case with an actual difference of about 10% for both programs. We also show the amount of stalls experienced by the benchmark as a fraction of total time. At larger heap sizes, there are no stalls. As the heap size is reduced, there will come a point when the collector cannot keep up and the mutator's allocation request will fail. For the allocation-intensive Deque benchmarks, the failure point occurs at $1.43 \times m$. Our predicted N_{min} value of 1.457 is correctly above the actual failure points.

Because the average collection time includes multiple phases of a program, it can be significantly lower than the maximum collection time. We see that the gap between T_{max} and collection time shrinks from 10% to about 2% and 6% when one considers maximum rather than average collection time. For space, N_{min} has only a worst-case flavor as there is adequate heap space only if the heap is sufficient at every collection. The space bound is within 3% of when stalls begin. Our time and space bounds are not only empirically validated but are tight.

In general, time spent for a single collection falls as the heap size is decreased since the sweep phase will take less time. It may seem surprising that this happens even when the heap size is taken below N_{min}. However, falling below this safe point causes mutator stalls but does not penalize the collector at all. In fact, because the mutator is stalled, it can no longer interfere with the collector which will additionally, though very slightly, speed up collection. Of course, since the overall goal is to avoid mutator stalls, operating in this regime is inadvisable.

Figure 7. Throughput measurements for Deque. (a) Execution duration in cycles of Deque; (b) execution time in milliseconds of Deque.



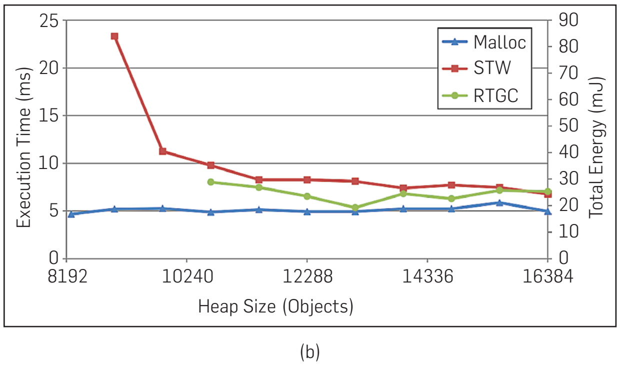
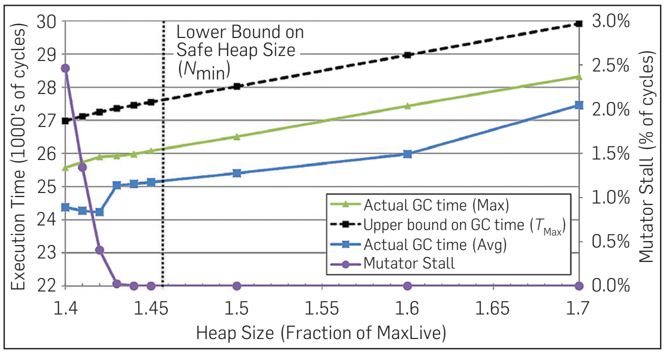


Figure 8. Comparison of predicted to actual duration and space usage.



8. RELATED WORK

We provide only a brief summary of related work and refer the reader to our full paper[6] for more details and citations.

There has been very little work on supporting high-level memory abstractions in reconfigurable hardware, and none on garbage collection. Simsa and Singh[14] have explored compilation of C subprograms that use malloc/free into VHDL or Bluespec. LEAP scratchpads[1] provide an expandable memory abstraction which presents a BRAM interface, but uses off-chip RAM if the structure is too large to fit, and transparently uses the on-chip BRAM as a cache. Such a system could be coupled with ours in order to provide a larger, virtualized memory, albeit at the expense of determinism and throughput.

Meyer[12] has built a special-purpose processor and an associated garbage collection co-processor on an Altera APEX FPGA. However, the design and the goals were very different. Meyer's collector is for a general-purpose heap allocated in DRAM, and for a program operating on what is for the most part a conventional CPU. The collector is implemented with a microcoded co-processor, and the general CPU is modified with a special pointer-related support. Because of this software-in-hardware approach, pauses can reach 500 cycles. In contrast, our approach is a fully custom logic that is much more tightly integrated with the memory, for "programs" that are also synthesized into hardware, and with deterministic single-cycle memory access. This allows us to attain zero pauses.

There are also garbage-collected systems[9, 13] in which the FPGA participates in a garbage collected heap but performs no collection itself. Yet others have designed special-purpose ASICs that perform a custom collection algorithm or provide a set of instructions to accelerate a more general class of algorithms. These are fundamentally different because the heap is on the CPU and not the FPGA.

9. CONCLUSION

We have described our design, implementation, and evaluation of the first garbage collectors to be completely synthesized into hardware. The real-time version causes *zero* cycles of interference with the mutator.

Careful implementation allows a closed-form analytic solution for worst-case execution time (WCET) of the collector, and a lower bound on heap size to achieve real-time behavior. These bounds are also cycle-accurate.

In software there are large trade-offs between stop-the-world and real-time collection in terms of throughput, latency, and space. Our measurements show that in hardware the real-time collector is faster, has lower (zero) latency, and can run effectively in less space.

This performance and determinism is not without cost: our collector only supports a single fixed object layout. Supporting larger objects with more pointers is a relatively straightforward extension of our design; supporting multiple object layouts is more challenging, but we believe can be achieved without sacrificing the fundamental advantages.

Garbage collection of programs synthesized to hardware is practical and realizable!

Acknowledgments

We thank the members of the Liquid Metal team at IBM Research who contributed insights, infrastructure, and a stimulating working environment: Joshua Auerbach, Ioana Baldini, Stephen Fink, and Rodric Rabbah.

References

1. Adler, M. et al. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceeding of International Symposium on Field Programmable Gate Arrays* (2011), 25–28.
2. Auerbach, J., Bacon, D.F., Cheng, P., Grove, D., Biron, B., Gracie, C., McCloskey, B., Micic, A., Sciampacone, R. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM International Conference on Embedded Software* (2008), 245–254.
3. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of ACM International Conference on Object Oriented Programming Systems, Languages, and Applications* (Oct. 2010), 89–108.
4. Bachrach, J., Huy Vo, B.R., Lee, Y., Waterman, A., Avidienis, R., Wawrzynek, J., Asanovic, K. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC)* (Jun. 2012), 1212–1221.
5. Bacon, D.F., Cheng, P., Rajan, V.T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003), 285–298.
6. Bacon, D.F., Cheng, P., Shukla, S. And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation* (Jun. 2012), 23–34.
7. Blelloch, G.E., Cheng, P. On bounding time and space for multiprocessor garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Atlanta, Georgia, Jun. 1999), 104–117.
8. Cook, B. et al. Finding heap-bounds for hardware synthesis. In *Formal Methods in Computer-Aided Design* (Nov. 2009), 205–212.
9. Faes, P., Christiaens, M., Buytaert, D., Stroobandt, D. FPGA-aware garbage collection in Java. In *proceedings of the IEEE International Conference on Field Programmable Logic and Applications* (FPL 2005), 675–680.
10. Greaves, D., Singh S.. Kiwi: Synthesis of FPGA circuits from parallel programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (2008).
11. Henriksson, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology (July 1998).
12. Meyer, M. An on-chip garbage collection coprocessor for embedded real-time systems. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time ComputingSystems and Applications* (2005), 517–524.
13. Schmidt, W.J., Nilsen, K.D. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages andOperating Systems* (1994), 76–85.
14. Simsa, J., Singh, S. Designing hardware with dynamic memory abstraction. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (2010), 69–72.
15. Xilinx. Virtex-5 family overview. Technical Report DS100, Feb. 2009.
16. Yuasa, T. Real-time garbage collection on general-purpose machines. *J. Syst. Software 11*, 3 (Mar 1990), 181–198.

David F. Bacon, Perry Cheng, Sunil Shukla ([bacon, perry, skshukla]@us.ibm.com), IBM T.J. Watson Research Center, NY.

© 2013 ACM 0001-0782/13/12 $15.00