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

David F. Bacon Perry Cheng Sunil Shukla
IBM Research
{dfb,perry,skshukla}@us.ibm.com

Abstract

Programmers are turning to radical architectures such as reconfigurable hardware (FPGAs) to achieve performance. But such systems, programmed at a very low level in languages with impoverished abstractions, are orders of magnitude more complex to use than conventional CPUs. The continued exponential increase in transistors, combined with the desire to implement ever more sophisticated algorithms, makes it imperative that such systems be programmed at much higher levels of abstraction. One of the fundamental high-level language features is automatic memory management in the form of garbage collection.

We present the first implementation of a complete garbage collector in hardware (as opposed to previous "hardware-assist" techniques), using an FPGA and its on-chip memory. Using a completely concurrent snapshot algorithm, it provides single-cycle access to the heap, and never stalls the mutator for even a single cycle, achieving a deterministic mutator utilization (MMU) of 100%.

We have synthesized the collector to hardware and show that it never consumes more than 1% of the logic resources of a high-end FPGA. For comparison we also implemented explicit (malloc/free) memory management, and show that real-time collection is about 4% to 17% slower than malloc, with comparable energy consumption. Surprisingly, in hardware real-time collection is superior to stop-the-world collection on every performance axis, and even for stressful micro-benchmarks can achieve 100% MMU with heaps as small as 1.01 to 1.4 times the absolute minimum.

Categories and Subject Descriptors B.3.3 [*Memory Structures*]: Worst-case analysis; B.5.1 [*Register-Transfer-Level Implementation*]: Memory design; B.7.1 [*Integrated Circuits*]: Gate arrays; C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Memory management (garbage collection)

General Terms Design, Languages, Experimentation, Performance

Keywords Block RAM, FPGA, High Level Synthesis, Garbage Collection, Real Time

1. Introduction

The end of frequency scaling has driven architects and developers to parallelism in search of performance. However, general-purpose MIMD parallelism is 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 general-purpose computing platform of all is reconfigurable hardware, in the form of Field-Programmable Gate Arrays (FPGAs).

FPGAs are now available with over 1 million programmable logic cells and 8 MB of on-chip "block RAM", providing a massive amount of bit-level parallelism combined with single-cycle access to the memory capacity of a VAX-11/780. Furthermore, because that memory is distributed throughout the chip in 18 Kbit units, algorithms with huge data bandwidth can also be implemented.

However, programming methodology for FPGAs has lagged far behind their capacity, which in turn has greatly reduced their application to general-purpose computing. The most common languages for FPGA programming 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, in particular, the Kiwi project [14] which uses C#, and the Liquid Metal project, which has developed the Lime language [4] based on Java.

However, up until now, whether programmers are writing in low-level HDLs or high-level languages like Kiwi and Lime, use of dynamic memory management has only just begun to be explored [11, 25], and use of garbage collection has been nonexistent.

In this paper we present a garbage collector synthesized entirely into hardware, capable of collecting a heap of *uniform* objects completely concurrently. We call such a heap of uniform objects a *miniheap*. By uniform we mean that the shape of the objects (the size of the data fields and the location of pointers) is fixed. Thus we trade a degree of flexibility in the memory layout for large gains in collector performance.

In the FPGA domain this 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.

For instance, the Lime language provides the capability to map a graph of compute tasks, each with its own private heap memory, onto the FPGA. Thus it is more important to handle many small heaps with a high degree of efficiency, than to handle a high degree of heterogeneity in the data layout.

Historically, it is also interesting to note that McCarthy's invention of garbage collection for LISP [18] was in the context of an IBM 704 with only 4K 36-bit words, essentially used as a fixed-format heap of S-expressions.

Algorithmically, our collector is a fairly straightforward Yuasa-style snapshot-at-the-beginning concurrent collector [35], with a linear sweep phase. However, 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, and the design can actually support multiple simultaneous memory operations per cycle. 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).

The collector we describe can be used either directly with programs hand-written in hardware description languages (which we explore this paper) or as part of a hardware "run-time system" targeted by a compiler for a C-to-gates [11, 25] or high-level language [4, 14] system including dynamic memory allocation. The latter is left to future work, and we concentrate in this paper on exploring the design, analysis, and limits of the hardware collector.

The contributions of this paper are:

- the first implementation of an entire garbage collector in hardware (as opposed to hardware-assist or microcode), including both stop-the-world and fully concurrent variants;
- the first garbage collector to entirely eliminate mutator interference by the collector (by even a single cycle), thereby achieving minimum mutator utilization (MMU [7]) of 100%;
- an implementation of the collector in Verilog along with demanding hardware-based client applications performing up to one memory allocation every 14 cycles;
- a performance evaluation showing the cost of garbage collection in absolute terms and relative to malloc/free, including dynamic measurement of throughput and energy;
- analytic closed-form worst-case bounds (in cycles) for collection time and minimum heap size required for 0-stall real-time behavior, along with experimental evidence of safety, and tightness within 2-6% for time and 3% for space; and
- an exploration of the design space showing that in hardware, real-time collection simultaneously achieves higher throughput, and lower latency, memory usage, and energy consumption than stop-the-world collection.

2. FPGA Background

Field Programmable Gate Arrays (FPGAs) are programmable logic devices consisting of 4- or 6-input look-up tables (LUTs) which can be used to implement combinational logic, and flip-flops which can be used to implement sequential logic. On the Xilinx FPGAs which we use in this work, several LUTs and flip-flops are combined together to form a unit called a *slice*, which is the standard unit in which resource consumption is reported for FPGAs (Altera FPGAs use a slightly different architecture).

FPGAs also include a clock distribution network for propagating a globally synchronized clock to allow for the use of conventional clocked digital logic. Our collector takes advantage of this global clock in a number of ways, in particular to implement an efficient single-cycle atomic root snapshot.

The FPGA also contains a large amount of configurable routing resources for connecting the slices, based on the data flow in the hardware description language program. The routing resources are used by the place-and-route (PAR) tool during hardware synthesis.

2.1 Memory Structures on FPGAs

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

The Xilinx Virtex-5 LX330T [32] device that we use in this paper (one of the largest in that family) has a BRAM capacity of 1.5 MB; the latest generation of Xilinx devices, the Virtex-7, have as much as 8 MB of BRAM.

A single BRAM in a Virtex-5 FPGA can store up to 36 Kilobits (Kb) of memory. An important feature of BRAM is that it can be organized in various form factors (analogous to word sizes on a CPU). On the Virtex-5, form factors of 1, 2, 4, 9, 18, 36, 72, and so on are supported. A 36 Kb BRAM can also be used as two logically separate 18 Kb BRAMs. Moreover, a larger memory structure can be built by cascading multiple BRAMs horizontally, vertically or in a hybrid manner. Any memory structure in the design which is smaller than 18 Kb 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 either read, write, read-before-write, or read-after-write operations. Our collector makes significant use of read-before-write for things like the Yuasa-style write barrier [35].

BRAMs can also be configured for use as FIFO queues 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 tradeoffs qualitatively. Some tradeoffs 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. This is essentially the same design as the "big bag of pages" (BIBOP) style in conventional software memory allocator design, in which the metadata for the objects is implicit in the page in which they reside [28].

The fixed shape of the miniheaps is a natural match to the FPGA, which must use fixed-width data paths to transmit the objects and fields across the routing network to achieve high performance.

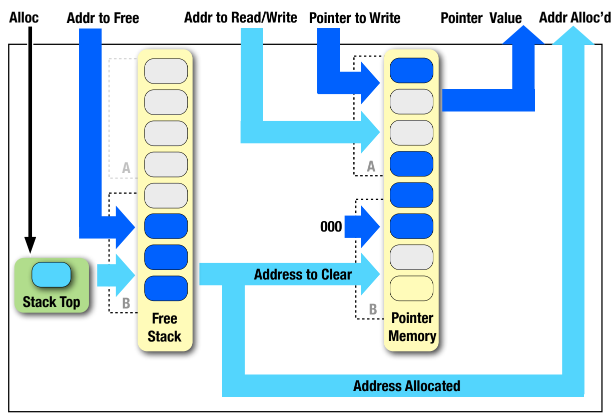


Figure 1. Memory module design for malloc/free interface, showing a single field of pointer type (for heap size $N = 8$ and pointer width $\log N = 3$). Block RAMs are in yellow, with the dual ports (A/B) shown. For each port, the data line is above and the address select line is below. Ovals designate 3-bit wide pointer fields; those in blue are in use.

3.1 Miniheap Interface

Each miniheap has an interface allowing objects to be allocated (and freed when using explicit memory management), and operations allowing individual data fields to be read or written.

In this paper we will consider miniheaps with one or two pointer fields and one or two data fields. This is sufficient for implementing many stack, list, queue, and tree data structures, as well as S-expressions. 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 (and sometimes power) 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 must 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.

For an object with f fields, there would be f BRAM sets with associated interfaces for the write and read values (but *not* an additional address port). And of course 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 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 dual-porting the BRAMs, can allow a read or write to proceed in parallel with an allocate or free.

3.3 Fragmentation and Other Trade-Offs

The BRAM quantization described in Section 2.1 can play a significant role in the efficiency of BRAM utilization. For instance, for a miniheap of size $N = 256$, pointers are 8 bits wide, so a single 18 Kb BRAM configured as 9 bits wide would be used. This wastes 1 bit per entry, but also wastes 1.75K entries, since only 256 (0.25K) entries are needed. The 1 bit wasted per field is a form of *internal fragmentation* and the 1.75K wasted fields are a form of *external fragmentation*.

To reduce external fragmentation, multiple fields of the same size could be implemented with a single BRAM set. However, since BRAMs are dual-ported, supporting more than two fields would result in a loss of parallelism in terms of field access. Furthermore, since we use one BRAM port for initialization of fields when they are allocated, this effect comes into play even with two fields.

The opposite approach is also possible: multiple fields can be implemented with a single BRAM set, resulting in a wider data width. In principle this can reduce internal fragmentation. However, in practice we find that this can actually result in poorer resource allocation. A wider data width also means that updates to individual fields must be performed with a read/modify/write sequence, which requires two cycles. Furthermore, the read/modify/write can not be pipelined, so in addition to higher latency, throughput can be halved.

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 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 can not be performed in the same cycle as `malloc` or `free`, and the latter require two cycles rather than one.

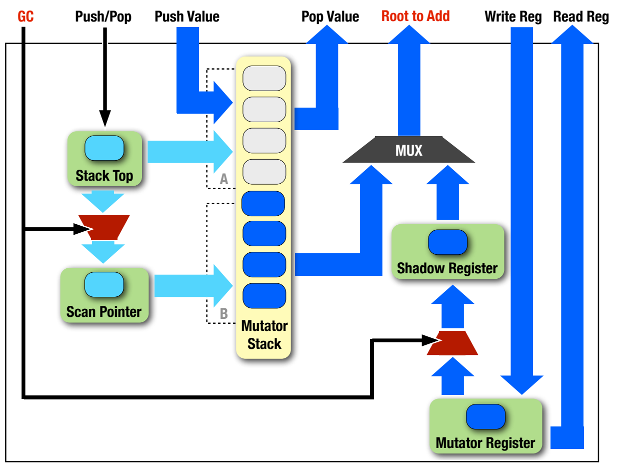


Figure 2. Single-Cycle Atomic Root Snapshot Engine

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 architecture of these two styles of collector are radically different. In hardware, the differences are much smaller, as the same fundamental structures and interfaces are used.

The concurrent collector has a few extra data structures (implemented with BRAMs) and also requires more careful allocation of BRAM ports to avoid contention, but these features do not negatively affect the use of the design in 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 collector 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 [35] 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 incorporating 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 [3, 22]).

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

This is in contrast to the "incremental update" style algorithms of Steele [27] and Dijkstra [12] (and numerous successors), which attempt to "chase" objects that are freshly allocated during collection.

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.

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 can not 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.

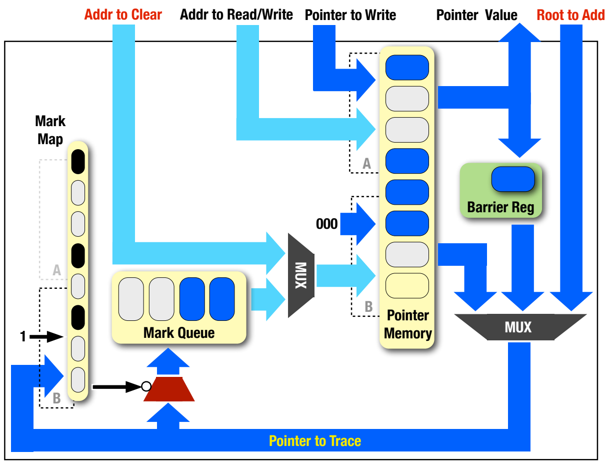


Figure 3. Tracing Engine and a Single Pointer Memory

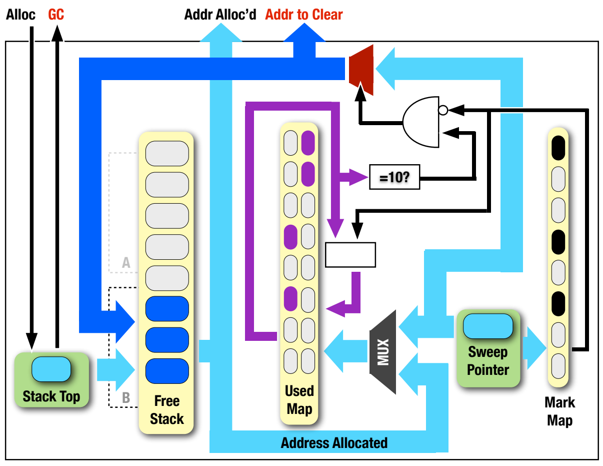


Figure 4. Free Stack and Sweeping Engine

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 over-written 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). The Mark Queue is a BRAM used in FIFO (rather than random access) mode.

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

4.3.1 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.

Furthermore, the two mark queues are MUXed together and 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 $3N/8 + R$ (where R is the maximum number of roots), which guarantees that the queues will never overflow. For a formal argument, see Appendix A.

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.

The final optimization is that since there are now two write barrier registers and two mark queues, the write barrier values are not processed until there are two of them. This means that the mark engines can make progress every other cycle even if the application is performing one write per cycle.

4.3.2 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.

We are currently investigating speculative "pointer forwarding" optimizations but they have not been implemented so we merely note that it may be possible to remove at least one if not both of these bubbles at a modest cost in additional logic.

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 machine cycle. Sweeping is a simple linear scan. The Sweep Pointer is initialized to 1 (since slot 0 is reserved for `null`), and on every cycle (except when pre-empted 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 is based on avoiding unneeded logic. 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.

5. Analysis of 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, 16], 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

$$T = T_R + T_M + T_W + T_X + T_S + T_A \tag{1}$$

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. Pre-emption 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 pre-empts 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

$$T_{\text{worst}} = R + 5N + 5 \tag{2}$$

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, e.g. 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

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

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). Obviously 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

$$N_{\min} = m + 2\alpha T_{\max} \tag{4}$$

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

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

6. Experimental Methodology

Since we have implemented the first collector of this kind, we can not simply use a standard set of benchmarks to evaluate it. Therefore, we have implemented two micro-benchmarks intended to be representative of the types of structures that might be used in an FPGA: a doubly-ended queue (deque), which is common in packet processing, and a binary tree, which is common for algorithms like compression.

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 there are no higher order effects, 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*.

There are many FPGA product lines and many different configurations within each line. Our experiments are performed using a Xilinx Virtex-5 LX330T [32], which is the largest chip within the Virtex5 LXT product line (that is now two generations old). Given that the motivation to use dynamic memory management will go up with complexity, and complexity will go up with larger chips, we believe that this is a good representative design.

The LX330T has 51,840 slices and 11,664 Kb (1.4 MB) of Block RAM. Fabricated in 65nm technology, the chip is theoretically capable of being clocked at up to 550 MHz, but realistic designs generally run between 100 and 300 MHz.

For each design point, we perform complete synthesis, including place-and-route (PAR), for the LX330T chip. PAR is the final physical synthesis stage and it reports the highest clock frequency the design can be clocked at, as well as the device resource utilization such as slices and BRAM. We used Xilinx ISE 13.4 tool for synthesis.

6.1 Description of Benchmarks

Our first benchmark is a binary search tree which is standard member of a family of binary tree data structures including variants like red-black trees, splay trees, and heaps. Though all the standard operations are implemented, the benchmark, for simplicity, exports only three operations: insert, delete, and traverse. The benchmark can be run against a workload containing a sequence of such operations. Our workload generator is configured to keep the maximum number of live nodes to 8192 while bursts of inserts and deletes can cause the instantaneous amount of live nodes to fall to $7/8$ of that. The burstiness of the benchmark necessitates measuring the allocation rate dynamically through instrumentation but provides a more realistic and challenging test for our collector. Traversal operations are included to confirm that our collector is not corrupting any data as the heap size is reduced. The allocation rate of the binary tree is proportional to the tree depth and could be characterized as intermediate for micro-benchmarks. In the context of a complete program, the final allocation rate is potentially even lower. As far as pipeline stalls, there will only be $B = 3$ stalls since the fanout of the tree fills the pipeline after the first two levels have been processed.

The second benchmark is a deque (double-ended queue). The doubly-linked list can be modified by pushes and pops to either the front or back. As before, our workload consists of a random sequence of such operations while keeping the maximum amount of live data to 8192. To contrast with the previous benchmark, there are no deliberate bursts which makes the allocation rate more consistent but also keeps the amount of live data always quite close to the maximum. Because there is no traversal or computation, the allocation rate is much higher and stresses the collector much more. As far as traversal, a doubly-linked list is equivalent to two singly-linked lists each of half the length. Thus, there will be $B = m/2$ stalls in the marking pipeline.

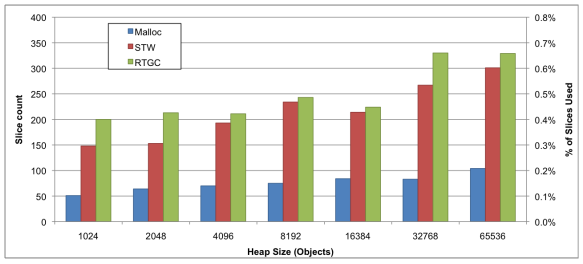


Figure 5. FPGA Logic Resource (slice) Usage

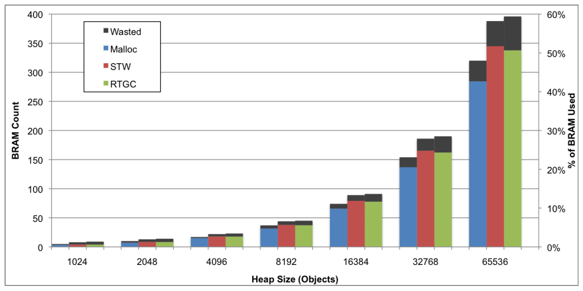


Figure 6. Block RAM Usage, including fragmentation wastage

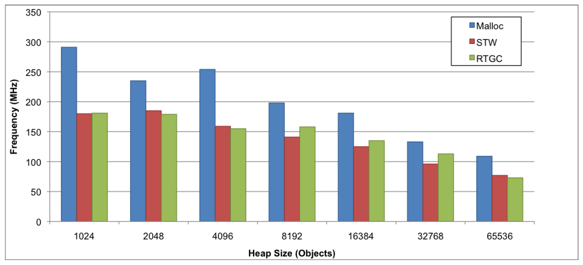
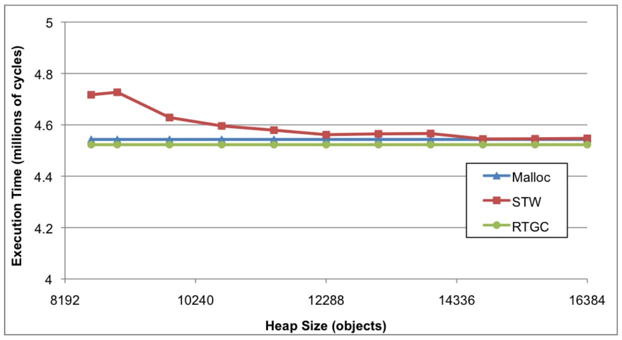


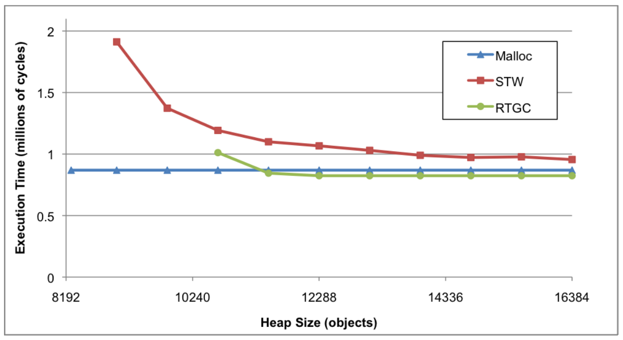
Figure 7. Synthesized Clock Frequency

7. Evaluation

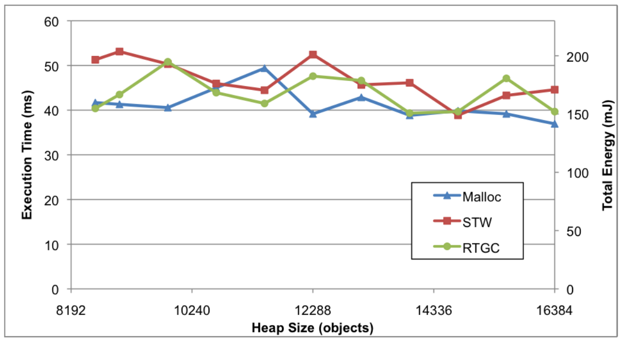
We begin by examining the cost, in terms of static resources, of the 3 memory managers – malloc/free ("Malloc") , stop-the-world collection ("STW"), and real-time concurrent collection ("RTGC").



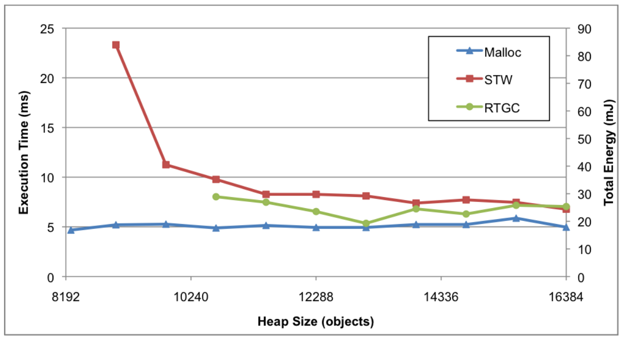
(a) Execution duration in cycles of Binary Tree



(b) Execution duration in cycles of Deque



(c) Execution time in milliseconds of Binary Tree



(d) Execution time in milliseconds of Deque

Figure 8. Throughput measurements for the Binary Tree and Deque Microbenchmarks. Because energy consumption is dominated by static power, which is virtually constant, graphs (c) and (d) also show energy in millijoules; the curves are identical.

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. The results are shown in Figures 5 to 7. Figure 5 shows the utilization of non-memory logic resources (in slices). As expected, garbage collection requires more logic than Malloc. Between the two collectors, RTGC requires between 4% to 39% more slices than STW. While RTGC consumes up to 4 times more slices than Malloc in relative terms, in absolute terms it uses less than 0.7% of the total slices even for the largest heap size so logic consumption for all 3 schemes is effectively a non-issue.

Figure 6 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.

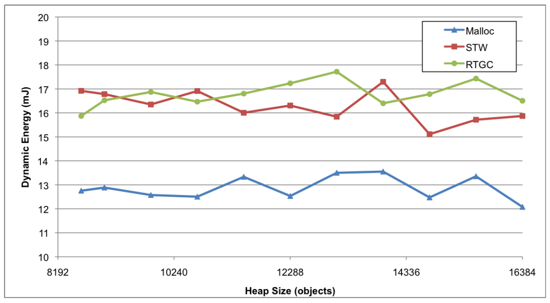
Finally, Figure 7 shows the synthesized clock frequency at different design points. Here we see a significant effect from the more complex logic for garbage collection: even though it consumes relatively little area, clock frequency for garbage collection is noticeably slower (15-39%) than Malloc across 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 Dynamic Measurements

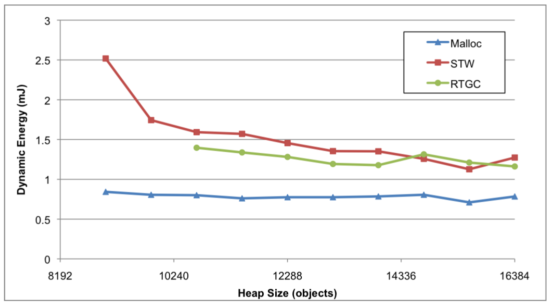
So far we have discussed the costs of memory management in the absence of applications; we now consider what happens when the memory manager is “linked” to the microbenchmarks from Section 6.1. Unlike the previous section, where we concentrated on the effects of a wide range of memory sizes on static 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.

7.1.1 Throughput

Figure 8 shows the throughput of the benchmarks 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



(a) Dynamic Energy in milli-joules of Binary Tree



(b) Dynamic Energy in milli-joules of Deque

Figure 9. Dynamic Energy

duration (graphs (a) and (b)), but also, since the synthesizable clock frequencies vary, in physical time (graphs (c) and (d)).

The Binary Tree benchmark goes through phases that are allocation- and mutation-intensive, and those that are not. As a result its allocation rate α is 0.009 objects/cycle, and its mutation rate μ is 0.02 pointer writes/cycle, when considered over a window size of m cycles. Because of these relatively low rates, the duration in cycles in Figure 8(a) of both Malloc and RTGC stays constant from $2m$ all the way down to $1.1m$. RTGC actually consumes slightly fewer cycles since it does not need to issue explicit `free` operations. Because STW pauses the mutator, each collection increases the total number of cycles required. As the heap gets tight, the duration in cycles for STW rises quickly.

However, when we obtain the physical time by dividing total duration in cycles by synthesized clock frequency, as shown in Figure 8(c), things become less cut and dried. Although Malloc alone can be synthesized at considerably higher frequencies than STW or RTGC (Figure 7), it is often the application rather than the memory manager that becomes the limiting factor on clock speed. Therefore, though the differences between the three memory managers is small, there are many chaotic variations. These are primarily explained by expected random variations in the synthesis tool which uses simulated annealing to achieve a good placement and routing. To maintain fairness, we have used same settings for all the synthesis jobs and avoided explicit manual optimizations.

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 8(b), even at heap size $N = 2m$, STW consumes noticeably more cycles, rising to almost double the cycles 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 cannot keep up with the mutator.

The Deque benchmark is considerably simpler than Binary Tree in terms of logic, so it has a correspondingly lower impact on synthesized clock frequency. The effect is seen clearly in Figure 8(d): 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 of course does not interrupt the application at all.

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 tradeoff in software.

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 considerably more palatable. As predicted, this performance gap only decreases with more complex applications.

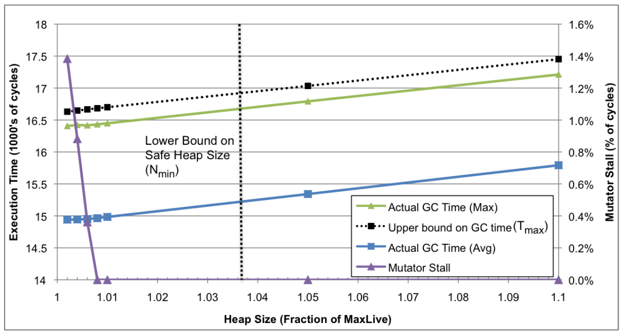
7.1.2 Energy

The *energy* (or power-delay product) is the product of average power dissipation and physical time. It is a better metric than power alone since it takes into the account the actual time needed to finish a given workload. For example, dynamic power consumption can be reduced by lowering the clock frequency but it does not mean that the energy consumed to finish the workload is also reduced. To calculate the energy we need to have knowledge of the power consumption and the total time taken to finish the workload. We calculated the power, which is a sum of static and dynamic power, using the Xpower tool from Xilinx [33]. The Xpower tool provides an approximate estimate of the power consumed by a design, for a particular FPGA device, by taking several factors into account such as simulation data, clock frequency, supply voltages, ambient and junction temperature. The accuracy of the result depends most importantly on the simulation data which is a representative of the target application. Based on the switching activity of the nets embedded in the simulation data, the dynamic power can be calculated very accurately. Static power consumption (the power consumed by the chip simply when it is on) for an FPGA chip is independent of the design. We have found that the static power contributes more than 90% of the total power for all the cases we investigated.

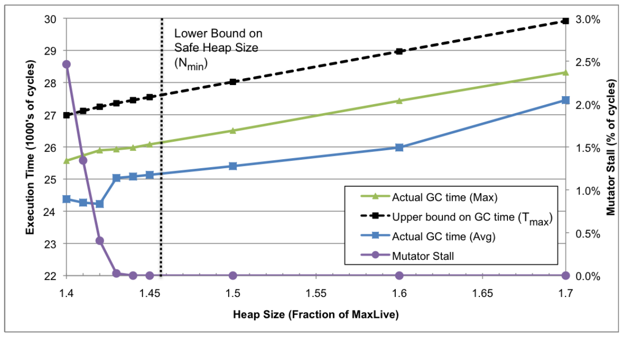
We synthesized the Binary Tree and Deque benchmarks for each heap sizes shown in Figure 8 to get the clock frequency and generate a fully placed-and-routed netlist. Then we simulated each design using the ModelSim simulator to generate the toggle activity of each net in the design and the result is stored in the VCD format [19]. We fed the synthesized netlist, the VCD file and the clock frequency constraint to the Xpower tool which provides detailed information about the static and dynamic power consumption. The power consumption is then multiplied by the execution time (in seconds) to calculate the total energy consumption.

The total energy consumption for Binary Tree and Deque is shown in Figure 8(c) and Figure 8(d) respectively (using the secondary right-hand axis). Because static power is so much larger than the dynamic power, the total power tracks the execution time within 1% and it is shown on the same graph. To demonstrate the differences in energy due to the design and benchmarks, the dynamic energy consumption for the Binary Tree and Deque benchmarks is shown in Figure 9.

For the Binary Tree benchmark, the average energy consumption (averaged over all the design points) for RTGC is lower than



(a) Binary Tree



(b) Deque

Figure 10. Comparison of analytic worst-case duration (in cycles) and heap space for RTGC compared to measured values

STW by 6% and higher than Malloc by 8%. For the Deque benchmark, on average RTGC consumes 14% less and 34% more energy than STW and Malloc respectively. The dynamic energy provides additional insight into the nature of benchmarks. For Binary Tree, RTGC consumes 3% and 30% more dynamic energy than STW and Malloc respectively. For Deque, RTGC consumes 8% less and 63% more dynamic energy than STW and Malloc respectively.

The analysis shows that the energy consumption is highly application-dependent. For both the benchmarks we considered it is safe to say that RTGC is a better choice than STW as far as energy consumption is considered. The average total energy consumption of Malloc is smaller than RTGC for both the benchmarks. However, as the complexity and size of benchmark increases the energy consumption gap between RTGC and Malloc diminishes.

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 10 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 be blocked. For Binary Tree this occurs when the heap is a mere $1.01 \times m$ while the more allocation-intensive Deque fails at $1.43 \times m$. Our predicted N_{min} values of 1.037 and 1.457 are 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.

8. Related Work

There has been very little work on supporting high-level memory abstractions in reconfigurable hardware, and none on garbage collection. Simsa, Singh, et al. [11, 25] have explored compilation of C subprograms that use malloc/free into VHDL or Bluespec for synthesis to FPGAs.

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.

Faes et al. [13] have built an "FPGA-aware" collector, with a completely different goal from ours: allowing the FPGA to maintain references into the CPU's main program heap. This facilitates co-processing by the FPGA.

8.1 Micro-coded Collectors

Meyer [20] has built a special-purpose processor and an associated garbage collection co-processor (using Baker's semi-space algorithm [6]), and realized them 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 CPU is modified with a special pointer register set, and pointer operations include support for read and write barriers. Some of the control logic also runs in software on the main CPU. By contrast, we have created 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.

Maximum collector pauses in Meyer's system are 500 cycles (2.5 μs at 200 MHz), due to root scanning. Read barriers and interlocks also cause pauses up to 200 cycles. By contrast our collector pauses for 0 clock cycles. Meyer's collector also requires heaps of 2 times the maximum live data in order to avoid non-real-time behavior, considerably more than our collector. Of course, our performance comes at a cost in flexibility: Meyer's collector handles arbitrary heap shapes and larger memories.

The Intel iAPX 432, the apotheosis of CISC architecture, did not provide explicit frees, and microcoded the marking portion of Dijkstra's on-the-fly collector (the rest was implemented in the iMAX-432 operating system). The system object table contained bits for the tri-color marking abstraction [31].

Schmidt and Nilsen [23] studied the design (using the DLX simulator) of a *garbage collected memory module* (GCMM), in which a memory module plugged into a conventional bus included paired DRAMs (for the collector semispaces) and an extra processor that ran Baker's algorithm [6]. The interface to the CPU for operations like allocation, root descriptors, and so on are implemented as memory mapped ports. They report delays of 0.5 ms to initiate a collection and 1 μs for allocate, fetch, and store.

8.2 Hardware-assisted Collection

Most hardware support for garbage collection has been in the form of specialized memory barriers (particularly read barriers) because of their high throughput cost. The Symbolics Lisp Machine [21] introduced this kind of hardware barrier to implement Baker's algorithm [6]. Both the Lisp Machine and SOAR [30] also introduced hardware support for generational write barriers.

Frustration with the availability and acceptance of custom processor designs to support garbage collection gave rise to a "stock hardware" line of research, using conventional CPU features to support collector operations [2, 8]. The pendulum swung back with the the Azul Vega processor and its associated collector [10]. Hardware support for read barriers, fast user-mode trap handlers (4-10 cycles), cooperative pre-emption, and special TLB support enabled a collector capable of handling very large heaps and large numbers of processors. The collector achieved an MMU of 21% at 50 ms (the authors hypothesize due largely to read barrier trap storms), and 52% at 200 ms. Ironically, the Azul collector has recently been ported to a "stock hardware" x86 platform running on a Linux kernel augmented with page remapping operations [29].

Hardware support has also been proposed for reference counting. Joao et al. [17] describe a processor extended with a "reference count coalescing buffer" which is able to filter over 96% of reference count traffic. Srisa-an and Lo [26] propose a co-processor that performs saturating reference counting (using 2- or 3-bit counts) backed by a conventional tracing collector. Yu [34] proposes a system with a reference-counted nursery and an incrementally traced mature space, using hardware support for allocation and write barriers and a specialized cache architecture.

Heil and Smith [15] apply hardware support for profiling to the garbage collection problem, by treating it as a form of instrumentation of store operations. The instrumented operations are then handled by service threads, and there is an additional thread to perform marking and sweeping.

Schoeberl [24] extends the JOP Java processor with hardware support for non-blocking object copies in the memory controller. This allows compaction operations to be pre-empted at the granularity of a single write.

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, achieving for the first time a minimum mutator utilization of 100%.

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. In addition, it consumes less overall energy because it can run to completion faster.

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 to our design; supporting multiple object layouts is more challenging but we believe can be achieved without sacrificing the fundamental advantages.

Compared to explicit memory management, hardware garbage collection still sacrifices some throughput in exchange for a higher level of abstraction. It may be possible to narrow this gap through more aggressive pipelining. However, the gap in space needed to achieve good performance is substantially smaller than in software.

For the first time, garbage collection of programs synthesized to hardware is practical and realizable.

A. Proof of Overflow Freedom

As described in Section 4.3.1, the queues for a pair of trace engines are each sized to have $3N/8 + R$ entries. Here we prove that neither queue will overflow. We begin by reviewing various GC operations that affect the queue. During the root phase, R roots are inserted into queue 0 while write barrier entries are inserted into queue 1. Thus, queue 1 can have up to R entries at the start of the tracing phase. Each trace step in the tracing phase pops from the fuller queue and then inserts up to 2 pointers (one in each queue). Interleaved with the traced steps are write barrier insertions which can also insert up to 2 pointers (one in each queue). If the write barrier step inserts only one pointer, then it will insert into the emptier queue.

We note that the operations in the mark phase are symmetric with respect to the two queues. Specifically, there is no bias in either the pushes and pops nor in the graph traversal since the role of left and right pointers are interchangeable. However, there is a difference in the root phase in that queue 0 is always equally or more full than queue 1. Thus, if there were to be overflow at all, it would occur in queue 0.

At the end of the root phase, the queues have up to R items so we must show that the queue will not grow by more than $3N/8$ during the tracing phase. The tracing phase can be considered to be a sequence of tracing steps and write barrier insertions. Tracing steps always pop an item and can push 0, 1, or 2 items. Write barrier insertions perform no pops but can push 0, 1, or 2 items. A write barrier operation that pushes 0 items does not change the state at all and need not be considered. However, a trace step that pushes 0 items cannot be so easily dismissed since the total number of trace steps is bounded by N. Thus, limiting any type of trace steps potentially affects the overall outcome.

We take full advantage of the balanced push and pop by noting that no operation increases queue imbalance beyond a difference of 1 between the queues since both the trace step pops from the fuller queue and the write barrier pushes onto the emptier queue for the case when it pushes only one item. If there were a pathological sequence of operations that causes either queue to overflow, then in the step just prior, the soon-to-overflow queue must be full and the other queue must be nearly full. Further, we need not consider write barrier operations that perform a single-push as we can conceptually rearrange them into half as many double-push operations which run faster. If there were to be an overflow, they would occur even faster with this rearrangement. Thus, we need only bound the total queue growth to $N/2$. We label the 3 types of tracing steps T_0, T_1, and T_2 based on the number of pushes they perform. Note that the net effect on the queue size is one less than the index. Similarly, we have W_2 which stands for the number of 2-push write barrier operations.

We are trying to maximize total occupancy which is given by

$$-T_0 + T_2 + 2W_2$$

subject to the usual non-negativity constraints as well as 3 additional constraints. The first constraint expresses the fact that we schedule at most one write barrier operation for each tracing operation. The second and third constraints require that the total number of pushes and pops cannot exceed the total number of objects.

$$\begin{aligned} T_0 + T_1 + T_2 &\geq W_2 \\ T_1 + 2T_2 + 2W_2 &\leq N \\ T_0 + T_1 + T_2 &\leq N \end{aligned}$$

With a slight rearrangement, these equations can be put in the "standard form" of a linear programming problem [9]. Although the quantities are constrained to be integral (making this in reality an integer linear programming problem), we are safe in droppping the integrality constraints as that only increases the feasible region. The over-approximation of the objective is not a soundness issue since we are establishing an upper bound. Conversely, the bound is also rather tight by noting the total size of the coefficients in the objective function.

The problem is easily solved with the simplex method and standard tableau techniques show that the problem is feasible and bounded with the objective maximized at $3N/4$ and the free variables T_0 and T_1 at zero while T_2 and W_2 are at $N/4$. Since the queues always maintain balance, we arrive at the final individual queue size by halving the $3N/4$ and including the capacity needed for the root phase to arrive at $3N/8 + R$.

We omit the actual tableaus as they are uninteresting and shed less insight than by examining a few key points in the space. From the objective function, it is intuitively desirable to maximize W_2. If we allow only W_2 and T_0 to be non-zero, then we will have both at $N/2$ with a total occupancy of $N/2$. Similarly, allowing only W_2 and T_1 into play at $N/3$ will achieve $2N/3$. Finally, W_2 and T_2 both at $N/4$ achieves the maximum of $3N/4$. If we were to leave out W_2 entirely, T_2 increases to $N/2$ but the objective actually decreases to $N/2$. The changes in these values confirm our intutition that trace operations that perform no pushes do not stress the queues and that maximizing the write barrier operations will cause the greatest occupancy.

Acknowledgments

We thank the members of the Liquid Metal team at IBM Research who contributed insights, infrastructure, and a stimulating working environment: Joshua Auerbach, Stephen Fink, and Rodric Rabbah. We also thank team members and the reviewers for their corrections and insightful comments which helped to improve the paper.

References

[1] M. Adler et al. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *FPGA*, pp. 25–28, 2011.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *PLDI*, pp. 11–20, June 1988.

[3] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In *EMSOFT*, pp. 245–254, 2008.

[4] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*, pp. 89–108, Oct. 2010.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL*, pp. 285–298, Jan. 2003.

[6] H. G. Baker. List processing in real-time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.

[7] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *PLDI*, pp. 104–117, June 1999.

[8] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP*, pp. 256–262, Aug. 1984.

[9] V. Chvatal. *Linear Programming*. W. H. Freeman and Company, 1983.

[10] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *VEE*, pp. 46–56, 2005.

[11] B. Cook et al. Finding heap-bounds for hardware synthesis. In *FMCAD*, pp. 205 –212, Nov. 2009.

[12] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[13] P. Faes, M. Christiaens, D. Buytaert, and D. Stroobandt. FPGA- aware garbage collection in Java. In *FPL*, pp. 675–680, 2005.

[14] D. Greaves and S. Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In *FCCM*, 2008.

[15] T. H. Heil and J. E. Smith. Concurrent garbage collection using hardware-assisted profiling. In *ISMM*, pp. 80–93, 2000.

[16] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[17] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *ISCA*, pp. 418–428, 2009.

[18] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, 1960.

[19] Mentor Graphics. *ModelSim SE Users Manual*. Version 10.0c.

[20] M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *RTCSA*, pp. 517–524, 2005.

[21] D. A. Moon. Garbage collection in a large LISP system. In *LFP*, Aug. 1984.

[22] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM*, pp. 159–172, 2007.

[23] W. J. Schmidt and K. D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS*, pp. 76–85, 1994.

[24] M. Schoeberl and W. Puffitsch. Nonblocking real-time garbage collection. *ACM Trans. Embedded Comput. Sys.*, 10:1–28, 2010.

[25] J. Simsa and S. Singh. Designing hardware with dynamic memory abstraction. In *FPGA*, pp. 69–72, 2010.

[26] W. Srisa-an, C.-T. D. Lo, and J. M. Chang. Active memory processor: A hardware garbage collector for real-time Java embedded devices. *IEEE Trans. Mob. Comput.*, 2(2):89–101, 2003.

[27] G. L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, Sept. 1975.

[28] G. L. Steele, Jr. Data representation in PDP-10 MACLISP. Tech. rep., MIT, 1977. AI Memo 420.

[29] G. Tene, B. Iyengar, and M. Wolf. C4: the continuously concurrent compacting collector. In *ISMM*, pp. 79–88, 2011.

[30] D. Ungar et al. Architecture of SOAR: Smalltalk on a RISC. In *ISCA*, pp. 188–197, 1984.

[31] Wikipedia. Intel iAPX 432, Nov. 2011.

[32] Xilinx. Virtex-5 family overview. Tech. Rep. DS100, Feb. 2009.

[33] Xilinx. Power methodology guide. Tech. Rep. DS786, Mar. 2011.

[34] W. S. Yu. Hardware concurrent garbage collection for object-oriented processor. Master's thesis, City University of Hong Kong, 2005.

[35] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Systems and Software*, 11(3):181–198, Mar. 1990.