A Stall-Free Real-Time Garbage Collector for FPGAs

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

ABSTRACT

Programmers are turning to diverse architectures such as reconfigurable hardware (FPGAs) to achieve performance. But such systems are far 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 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.

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 our collector is between 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.

This reprises work previously published in PLDI [2].

1. INTRODUCTION

FPGAs are now available with over 1 million programmable logic cells and 8 MB of on-chip block RAM, sufficient for complex computations without going off-chip. 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 abstractions are bits, arrays of bits, registers, wires, etc.

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 [4] which uses C#, and the Liquid Metal project, which has developed the Lime language [1] based on Java.

1.1 Background: Garbage Collection

Dynamic memory management provides a way of allocating parts of the memory to the application whenever requested and freeing the parts which are no longer in use. In explicit memory management, the programmer both allocates and frees the memory explicitly (`malloc`/`free` style). However, mistaken invocations of `free` can cause very subtle bugs, and omission causes storage leaks.

Garbage collection [6] is a form of automatic memory management where the collector reclaims the objects that are no longer in use, and `free` is no longer available to the programmer. Garbage collection works on the principle of pointer reachability. The reachability is determined through transitive closure on the object graph. It is assumed that an object in the heap which has no pointer reference to it from a reachable object or program variables (roots) can be safely reclaimed as a garbage and added to the pool of free objects.

Until now, whether programmers are writing in low-level HDLs or high-level languages like Kiwi and Lime, use of dynamic memory management on FPGA has only just begun to be explored, and then only in the context of explicit memory management [3, 7].

Garbage collection can be done either in a way which pauses the mutator (part of the application which creates objects and mutates the object graph) while collection is going on, known as stop-the-world ("STW") collection, or completely concurrently with the mutator, known as concurrent collection. Regardless of the scheme chosen for garbage collection, the aim is to keep the mutator pause time (due to collector activities) to a minimum. Real-time collectors ("RTGC") go beyond concurrency by also providing determinism (bounding the length of pauses and the distance between them). Concurrent and real-time collectors are significantly more complex because they must work correctly even though the object graph they are tracing is being modifed "out from under them". For a comprehensive overview, see [5].

1.2 Our Contribution

In this paper we present the first garbage collector synthesized entirely into hardware (both STW and RTGC variants), capable of collecting a heap of *uniform* objects. 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 (relative to what is common for software collectors) for large gains in collector performance.

Furthermore, the mutator has a 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.

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 [3, 7] or high-level language [1, 4] 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.

We also built an explicit (malloc/free) memory manager for comparison. We integrated the three memory managers with two benchmarks and compared the performance.

This paper is based on work previously published at PLDI 2012 [2].

2. 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

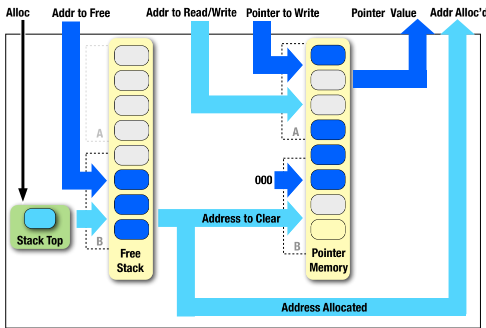


Figure 1: Memory module design for malloc/free interface, showing a single field of pointer type (for heap size $N = 8$ and pointer width $\log N = 3$). Block RAMs are in yellow, with the dual ports (A/B) shown. For each port, the data line is above and the address select line is below. Ovals designate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

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 5.

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 [8].

2.1 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 an arbitrary number of 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. Increasing the number of pointer fields is straightforward for malloc-style memory, but for garbage collected memory requires additional logic. We believe this is relatively straightforward to implement but the experimental results in this paper are confined to one- and two-pointer objects.

2.2 Miniheap with Malloc/Free

There are many ways in which the interface in Section 2.1 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 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. 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.

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.

3.1 Root Snapshot

The concurrent collector uses the snapshot-at-the-beginning algorithm. Yuasa's original algorithm [10] 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 a single stack and a single register.

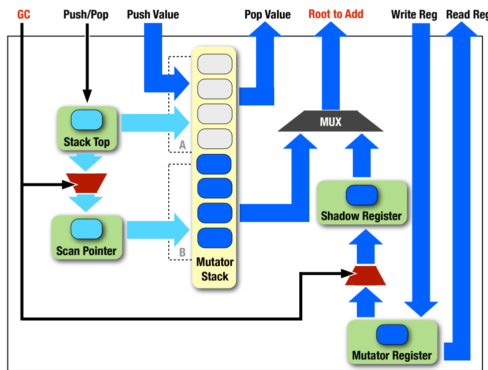


Figure 2: Single-Cycle Atomic Root Snapshot Engine

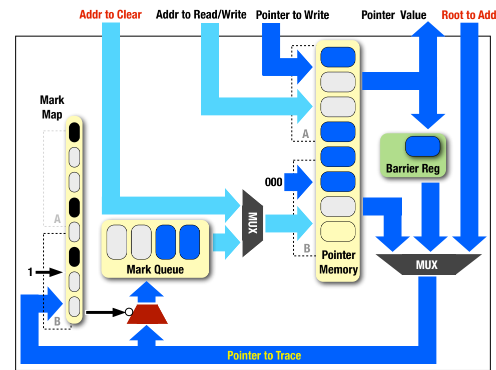


Figure 3: Tracing Engine and a Single Pointer Memory

The snapshot is controlled by the *GC* 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.

The register snapshot is obtained by using a shadow register. In the cycle after the *GC* signal goes high, the value of the mutator registers is copied into the shadow registers. 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 mutator 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.

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 3.3, 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.

3.2 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 (below).

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 non-null are fed to the MUX and thence back 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.

3.2.1 *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 queue 0 (the queue into which new roots are placed), unless it is empty in which case it is taken from queue 1. Using this design, we provision each of the 2 queues to be of size $N/2$, 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.

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.

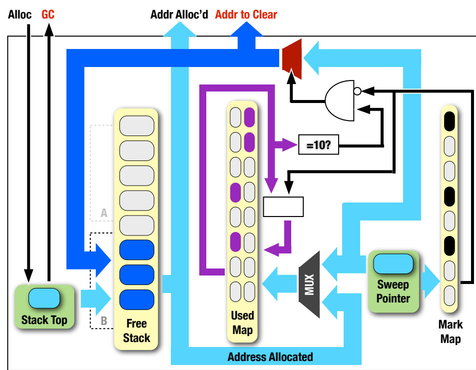


Figure 4: Free Stack and Sweeping Engine

3.3 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". 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 set to `00`. 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, note that 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, it is *not* marked. Thus our collector does not "allocate black". 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.

4. 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.

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 using a Xilinx Virtex-5 LX330T [9]. The LX330T has 51,840 slices and 11,664 Kb (1.4 MB) of Block RAM. For each design point, we perform complete synthesis, including place-and-route (PAR). We used Xilinx ISE 13.4 for synthesizing the designs.

4.1 Description of Benchmarks

Our first benchmark is a binary search tree which is standard member of a family of binary trees 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.

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.

5. EVALUATION

5.1 Static Measurements

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. The results are shown in Figures 5 to 7.

5.1.1 Logic usage

Figure 5 shows the utilization of 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

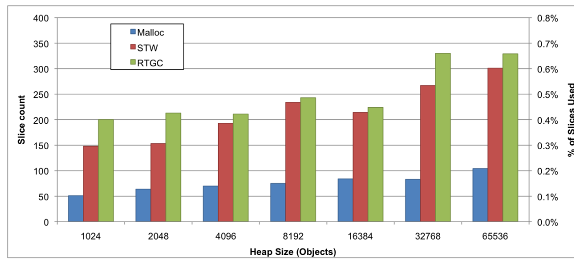


Figure 5: FPGA Logic Resource (slice) Usage

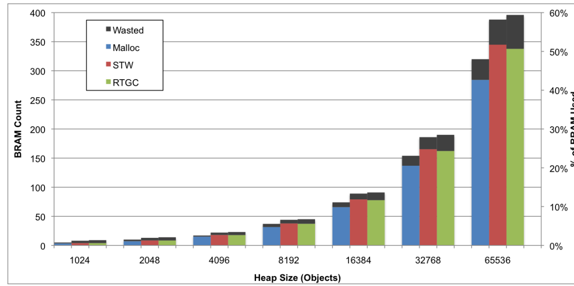


Figure 6: Block RAM Usage, including fragmentation wastage

0.7% of the total slices even for the largest heap size so logic consumption for all 3 schemes is effectively a non-issue.

5.1.2 *Memory usage*

Figure 6 shows BRAM consumption. 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.

5.1.3 *Clock frequency*

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.

5.2 Dynamic Measurements

So far we have discussed the costs of memory management in the absence of applications; we now turn to considering what happens when the memory manager is “linked” to the microbenchmarks from Section 4.1. 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 $2m$ by intervals of $1/10$ (including full synthesis at each design point),

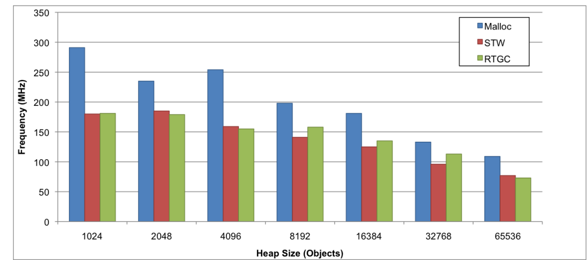


Figure 7: Synthesized Clock Frequency

to evaluate what happens to the memory managers as memory pressure varies from moderate to impossibly tight.

5.2.1 *Throughput*

Figure 8 shows the throughput of the benchmarks as the heap size varies for Malloc, STW, and RTGC. To fully understand the interaction of various effects, one must understand the throughput both in the duration in cycles (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.

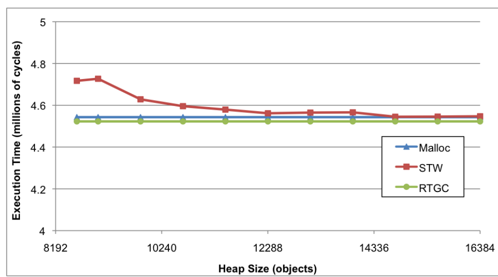
However, when we obtain the physical time by dividing total duration in cycles by synthesized clock frequency, as shown in Figure 8(a), things become less cut and dried. Although Malloc alone can be synthesized at considerably higher frequencies than STW or RTGC (Figure 7), it is often the application rather than the memory manager that becomes the limiting factor on clock speed. Therefore, the differences between the three memory managers is minimal and slightly chaotic due to variation in the synthesis tool.

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 can not keep up with the mutator.

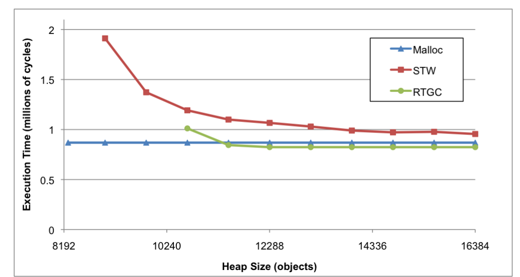
The Deque benchmark is considerably simpler than Binary Tree in terms of logic, so it has a correspondingly lower impact on synthesized clock frequency. The effect is seen clearly in Figure 8(d): Malloc synthesizes at a higher frequency, allowing it to make up RTGC’s slight advantage in cycles and consume 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 ever to use STW.

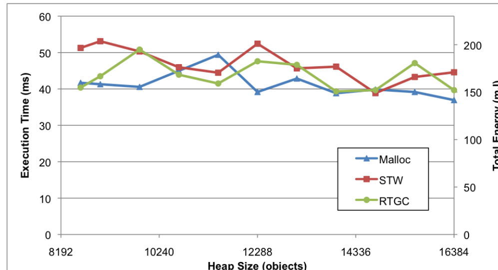
Furthermore, RTGC allows applications to run at far lower mul-



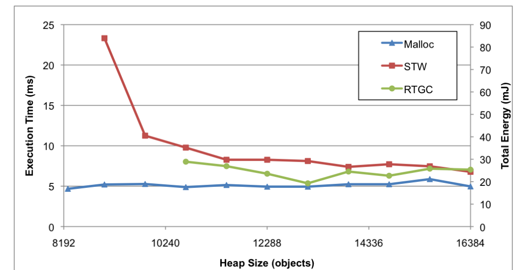
(a) Execution duration in cycles of Binary Tree



(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.

tiples 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.

5.2.2 *Energy*

Energy is a product of average power dissipation and physical time. The average power dissipation, as reported by the XPower tool from Xilinx, across all design points and memory managers remains almost constant and is dominated by the static power consumption which accounts for more than 90% of the total power. Hence the energy plot simply follows the physical time plot with scaling. The secondary vertical axis in Figure 8 (c) and (d) shows the energy consumption for binary tree and deque respectively.

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

6. CONCLUSION

We have described our design, implementation, and evaluation of the first garbage collectors to be completely synthesized into hardware. 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.

7. REFERENCES

[1] 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.

[2] D. F. Bacon, P. Cheng, and S. Shukla. And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. In *PLDI*, June 2012.

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

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

[5] R. Jones and R. Lins. *Garbage Collection*. John Wiley and Sons, 1996.

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

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

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

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

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