Direct Garbage Collection: Two-fold Speedup for Managed Language Embedded Systems

Rasmus Ulslev Pedersen

Department of Digitalization

Copenhagen Business School

Denmark

Martin Schoeberl

Department of Applied Mathematics and Computer Science

Technical University of Denmark

Abstract: More and more embedded systems are emerging based on managed language runtime systems using garbage collected languages such as Java, Python, or the .NET language family. Furthermore, the garbage collection (GC) process is a bottleneck in an embedded system, effectively blocking most other processes including mutator memory access, responding to inputs, or asserting outputs.

We demonstrate a valuable new heap memory architecture for garbage collected embedded systems, which works by creating a direct path between memory modules to achieve a two-fold speedup for a memory copy operation as compared to a baseline scenario using multiplexed shared address- and data-busses. This direct-path memory setup is generalizable, and memory modules will continue to work as expected when not engaged in garbage collection. The solution space is evaluated by simulating GC activity extracted from the Elephant Track GC tracer. One particular solution is also implemented in hardware to demonstrate the practical realization of the direct fast copy architecture.

Keywords: garbage collection; managed languages, embedded systems, realtime, memory, SRAM

Reference to this paper should be made as follows: Rasmus Ulslev Pedersen and Martin Schoeberl. (2017) 'Direct Garbage Collection: Two-fold Speedup for Managed Language Embedded Systems', *International Journal of Embedded Systems*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Rasmus Ulslev Pedersen works at Copenhagen Business School 2005 to present. He has given Ph.D.-level summer school classes on intelligent systems. At Department of Digitalization, he has taught IT, business intelligence, internet of things, and data mining. He is a co-founder of the Internet of Things Forum and IoTPeople. He has served on several EU FP7 projects.

Martin Schoeberl received his PhD from the Vienna University of Technology in 2005. From 2005 to 2010 he has been Assistant Professor at the Institute of Computer Engineering. He is now Associate Professor at the Technical University of Denmark. His research interest is on hard real-time systems, time-predictable computer architecture, and real-time Java. Martin Schoeberl has been involved in a number of national and international research projects: JEOPARD, CJ4ES, T-CREST, RTEMP, the TACLe COST action, and PREDICT. He has been the technical lead of the EC funded project T-CREST. He has more then 100 publications in peer reviewed journals, conferences, and books.

1 Introduction

As managed languages, such as Closure, JRuby, Jython, and Scala, that run on the Java Virtual Machine (JVM), become more and more prevalent (Li et al., 2013), the need to re-consider how garbage collection (GC) is done becomes evermore important. With the miniaturization of embedded systems and Cyber Physical Systems, i.e. *swarmlets* (Latronico et al., 2015) see Lee (2008), prototypes emerge based on Javascript/XML, which used to be a technology reserved for web clients. This class of systems also includes smart watches, drones, mobile devices, safety-critical systems (e.g. elevators) and industrial Internets of Things (IoT) (Hoske, 2015) for applications such as smart factory automation. The need

to invent even more efficient hardware solutions remains high. From the software side this discussion even includes ideas on embedding scripting languages (here brought in as a subcategory of managed languages) like Python to run on “bare metal" ARM processors, as discussed by Pedersen et al. (2009). Moreover, IoT sensors (Swan, 2012) enable companies to realize new business opportunities in logistics and healthcare (Fiedler and Meissner, 2013) based on micro information systems (Pedersen, 2010). The majority of these systems are battery-powered, and thus need to cope with a well-known energy challenge related to managed languages, since the frequent allocation and de-allocation associated with GC costs extra processor cycles. Ongoing research into solutions that address speed (e.g. Gomes et al. (2016)) and power consumption is particularly needed for the GC in managed languages such as Closure, JRuby, Jython, Scala, and Java itself (Sarimbekov et al., 2013). This paper investigates the possibility of designing fast, yet time-predictable hardware to further support the applicability of managed programming languages with automatic GC in embedded real-time systems. Indirectly, an additional benefit of *speed* is often reduced battery consumption, since the processor is able to *sleep* more. We call the proposed hardware solution for speeding up GC the **garbage collection logic (GCL)**.

The three main contributions of this paper are as follows: (1) an analysis of the requirements that GC places on real-time embedded systems, (2) the simulation and verification of two alternative hardware architectures, and (3) a prototype implementation on real hardware based on our open source software/hardware.

The paper is organized as follows: Section 2 describes related work and motivates the focus on the copy compaction process as a critical feature. Section 3 explains copying GC, outlines the solution space and introduces different architectures that address the realization of the copy operation in a baseline architecture and in the new direct copy GC architecture. Section 4 compares the solutions. In Section 5 we provide more details of the main solution, and in Section 6 we present a simulation based on a hardware description language. A proposed platform is implemented on a PCB and presented in . Section 8 gives a summary and evaluation of various aspects of the solutions. Section 9 provides a conclusion and indicates some valuable future directions for this work. All source code is available under

a simplified open source *BSD license*, and links to this material are given at the end of the paper.

2 Related Work

The focus there was on achieving GC with low blocking times to allow usage in interactive (real-time) applications. This is similar to our work, where we aim for a highly efficient GC to minimize blocking times due to the GC activity. A good introduction to garbage collection techniques can be found in the survey by Wilson (1994) and in Jones and Lins (1996).

The simplest way to avoid blocking times due to object copying is simply to avoid moving objects at all. The real-time GC of the JamaicaVM does exactly this (Siebert, 2002). Objects and arrays are split into fixed sized blocks and are never moved. This approach trades external fragmentation for internal fragmentation.

In a manner similar to the JamaicaVM approach, the Metronome GC splits arrays into small chunks called Arraylets (Bacon et al., 2003). Metronome compacts the heap to avoid fragmentation and the Arraylets reduce blocking time when copying large arrays. Both approaches, the JamaicaVM GC and Metronome, have to pay the price of a more complex (and time consuming) array access.

Another approach to allowing interruption of GC copy is to perform field writes to both copies of the object or array (Huelsbergen and Larus, 1993). This approach slows down write accesses, but these are less common than read accesses.

Nilsen and Schmidt propose hardware support, the object-space manager (OSM), for real-time GC on a standard RISC processor (Nilsen and Schmidt, 1992). The OSM redirects field access to the correct location for an object that is currently being copied. Nilsen and Schmidt's approach assumes that the hardware support is part of a (single) memory chip, whereas we propose a GC hardware that supports direct copy of objects between two standard memory chips.

Compared to GC with a fixed block size (e.g., the JamaicaVM approach), a compacting GC avoids internal fragmentation. Blackburn et al. (2004) describe some of the main forms of compacting GC. A generational collector distinguishes between a so-called *nursery* region, which is reserved for newly created objects, most of which *die young*. Later on in this paper, we focus on testing our solutions on the nursery region since this region will also benefit significantly from faster copying time. Some recent software for precise GC tracing named *Elephant Track* (Ricci et al., 2011) has been developed, which Li et al. (2013) and Sarimbekov et al. (2013) apply to analyse the typical size of objects as well as characteristics for generational GC using a nursery region.

The benefit related to compacting GC, and then also to generational GC, is the additional memory bandwidth that is consumed by the object copy and the handling of the object relation is avoided. Field and array accesses respectively need two and three memory accesses for an object layout with a handle indirection. In the JamaicaVM, the cost of the field and array access varies and depends on the location of the field or the layout of the array. In the average case, the cost of a field access is close to one and that of an array access close to two memory accesses (Siebert, 2000). With the jvm98 benchmarks SPEC (1998) executing on the JamaicaVM, most arrays are allocated contiguously instead of as a tree of fixed sized blocks. However, in the worst case a tree needs to be traversed and 2+d memory accesses are needed for a tree of depth d.

Hardware support can be used to avoid blocking mutator threads during object copy (Schoeberl and Puffitsch, 2008, 2010). This non-blocking copy unit can be interrupted by a thread with a higher priority than the GC thread. Afterwards, the copy unit resumes with the copy process when the GC thread is active again. The copy unit also redirects object field reads and writes to the correct part of a possibly partially copied object. A particular non-blocking copy unit has been evaluated by an implementation in the context of the Java processor JOP (Schoeberl, 2008). The resulting maximum blocking time due to the object copy is 120 ns (12 clock cycles at 100 MHz). In contrast to the presented *direct-path-copy*, the JOP GC copy unit is part of the processor and handles just a single memory.

Our proposal is also tightly integrated with the memory, but benefits from two memory chips that represent the source and target of an object copy.

Meyer presents a hardware implementation of Baker's read-barrier (Baker, 1978) in an object-based RISC processor (Meyer, 2006). It is stated that the cost of the read barrier is between 5 and 50 clock cycles. The resulting minimum mutator utilization (MMU) for a time quantum of 1 ms was measured to be 55%. Close interaction between the RISC pipeline and the GC coprocessor allow redirection for field access in the correct semispace with a concurrent object copy. It is not explicitly described in the paper when the GC coprocessor performs the object copy. We assume here that the memory copy is performed in parallel with the execution of the RISC pipeline. In that case, the GC unit *steals* memory bandwidth from the application thread.

Maas, Asanovic, and Kubiatowicz propose to add hardware support for GC *into* a standard processor (Maas et al., 2016). They argue that today's common usage of managed languages with GC in large server-side applications calls for the introduction of hardware support for GC within the CPU. We completely agree with this line of reasoning. However, we aim initially to support embedded systems and we add GC logic externally to a standard RISC processor, CISC CPUs, or softcore processors like JOP or Altera/Intel Nios II.

Bacon, Cheng, and Shukla present a GC implemented in an FPGA (Bacon et al., 2012). The GC supports uniform objects, which are objects of similar size and layout. The idea is that the GC is used in pipelined hardware design, where each pipeline stage has its own heap and GC. Furthermore, their implementation uses only on-chip memory inside the FPGA, which is usually quite small. In contrast to Bacon et al.'s work, we support arbitrary object sizes using external memories to speed up the GC copy process.

3 Garbage Collection: Copying

In this section we introduce the GC copy process and provide examples which motivate our design decisions. The focus is on the copy process for a generational GC.

To avoid fragmentation, GCs usually perform some form of compaction (Wilson, 1994). In our discussion, we assume a copying GC with two *semispaces*, hereafter called *fromspace* and *tospace*. We label the first concrete memory chip A and the second concrete memory chip, if present, B.

Figure 1 illustrates the GC process. The stack and static fields are traversed for potential live references, which point to objects on the heap (Pedersen and Schoeberl, 2006). These initial references form the *root set*. Further live references are identified by traversing the root set for references to other live objects, which subsequently are traversed for references to additional live objects.

In the figure, three objects are hosted in the $fromspace$. Objects 1 and 3 are *alive* (a live reference points to each) and can be accessed from a running program (i.e. a mutator thread). During GC, These two objects will be copied to semispace $tospace$. Then the memory area of semispace $fromspace$, that starts at address $StartAddrRamA$ and ends at $TopAddrRamA$, can be reclaimed and is guaranteed to be defragmented (it is empty). Note that $tospace$ was empty before the copy process and now holds the live objects. As only live objects are copied into $tospace$, this semispace is defragmented as well.

Object 1, with which we begin, is copied from semispace $fromspace$ to semispace $tospace$ in an implementation dependent manner. If the copy process were to be performed by a standard processor, the data would automatically be copied into the memory cache before being written back into the main memory of the other semispace. Therefore, the copy phase of the GC trashes all data cache content. This process involves more steps and increases the time the system either has to halt the mutator thread or monitor the allocation of new objects and access to objects that are being copied while blocking input/output operations. With our proposed direct copy unit (a) we avoid trashing the data cache, and yet (b) we speed up the copy process by a direct link between the two memory chips.

The GC copy process steps are (see Figure 1):

① The root set contains a reference to *live* Object 1

② Object 1 is located in semispace *fromspace* in A via the object reference table

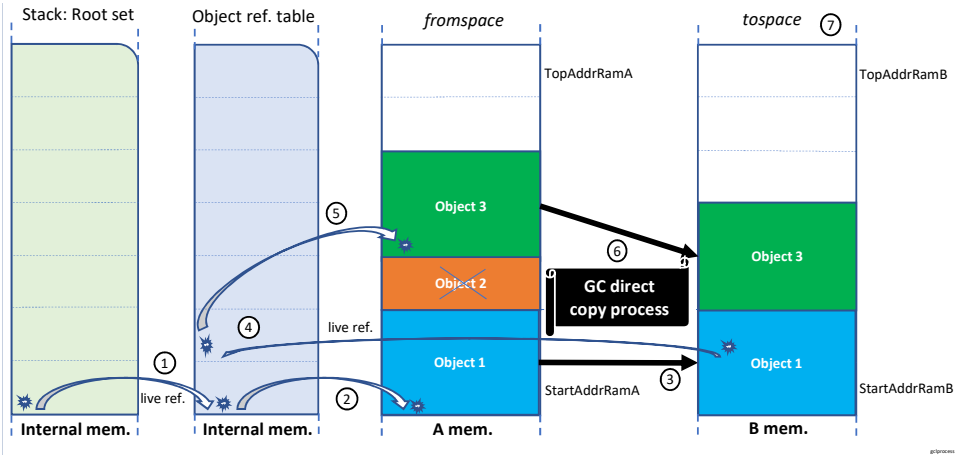

Figure 1: Garbage collection process

③ Object 1 is then copied to *tospace* in B

④ Object 3 is referenced by Object 1

⑤ Object 3 is also located via the object reference table

⑥ Object 3 is then copied to *tospace*

⑦ Finally *tospace* is *flipped* to become *fromspace* and vice-versa.

In our proposal the object copy is atomic. Therefore, we need to optimize this object copy process. However, the GC thread can be interrupted by a mutator thread after each copy. Therefore, the mutator thread(s) execute a snapshot-at-beginning write barrier (Yuasa, 1990) when writing into reference fields of an object. In this paper we assume that the GC thread has a period limited by its maximum possible period (similar to a paper by Schoeberl (2010)), which will not be the highest priority in the application.

The focus of this paper is on Steps 3 and 6: The object **copy** process. Here the per-object copy process is *blocking* and usually not thought to be interruptible. However, the direct copy process is not different from any other GC that would need to move objects. Here the process is just at least twice as fast since there is no on-chip data cache or buffer (see e.g., Figure 2, GCL $direct$) involved before the object is written to the destination address.

It is possible to implement the object reference table in different ways (cf. Chap. 30, *Formal Specification of the Object Memory* in relation to Smalltalk (Goldberg and Robson,

1983)). The fastest possible setup is when the object table is placed inside the GCL, using internal SRAM, since this is single-cycle read/write, and it does not require further databus sharing between memory components physically hosting the semispaces in memory A or B. Since the size of both A and B can be decided as the system is designed and implemented, it is also possible to use a larger SRAM for A than for B and reserve this space for the object table. The object reference table, also referred to as object handle table, is a useful construct as we work with managed languages, since the objects can be accessed without using direct pointers. Each object in the object table has a reference and a (physical or virtual) memory address pointed to. The memory address points to a word in the active semispace. When the object has been copied, the object reference table (see Figure 1) is updated accordingly. In the case that the object copying process can be done in a autonomous manner, the GC thread can update the object reference table concurrently.

Table 1 shows an extension of our former example. Assume that at some point we have had 9 different objects allocated in one semi-space on the heap. Three objects (object 1, object 2, and object 3) are already familiar from Figure 1. It is evident from the table below that only 4 of the 9 objects are alive, and that objects 0, 2, 4, 6, and 8 should be garbage collected.

Table 1: Copy objects marked alive from $fromspace$ to $tospace$

$fromspace$

Obj. ID	Addr	Size	Alive
0	0x000	0x10	0
1	0x010	0x20	1
2	0x030	0x30	0
3	0x060	0x40	1
4	0x0A0	0x50	0
5	0x0F0	0x60	1
6	0x150	0x70	0
7	0x1C0	0x80	1
8	0x240	0x90	0

$\Rightarrow$

$tospace$

Obj. ID	Addr	Size	Alive
1	0x000	0x20	1
3	0x020	0x40	1
5	0x060	0x60	1
7	0x0C0	0x80	1

After GC, i.e., copying the *live* objects to the other semispace, the memory will look as in Table1 (right side). The 4 copied objects are all of different sizes and if the copy process

were executed on a word-wide databus it would take 1 cycle/word to copy each object from $fromspace$ to $tospace$.

4 GC Copy Performance Analysis

This section discuss aspects of the design space w.r.t. GC performance. Our quantification includes databus width, operating frequency, and possible bus utilization. Figure 2 shows two different architectures for the GCL: (1) the GCL $shared$ and (2) the GCL $direct$ architectures.

4.1 GCL Architectures

The baseline architecture for a GC copying process is first to copy objects into a cache or small FIFO buffer in the processor and then write them back out to the *tospace* again. We call that the GCL $shared$ setup (see Figure 2). In other words, when there are live objects in $fromspace$ that need to be copied, for compaction, into $tospace$, one way to accomplish this is to read each word from $fromspace$ into the GCL and then write it back out to $tospace$. It is possible to refine this with a small FIFO queue (e.g. Figure 2 GCL $shared$).

We propose the following novel **garbage collection logic (GCL)** ***direct*** architecture that includes a direct connection between A and B, the memories hosting $tospace$ and $fromspace$, it is possible to get the two memories to read and write respectively to/from one another directly (e.g. Figure 2 GCL *direct*). It is novel since the authors have not found previous work where external memory ICs have been directly connected to stream live objects from one to the other before. It is valuable since this approach potentially frees the databus up for other work in concurrent systems. Logically and in implementations, the approach is to connect each corresponding data signal between the two memory ICs, A and B. So if for example each data signal is called DQ, the approach is to connect A_{DQ_1} to B_{DQ_1} and so forth until the whole databus is connected. Note that both memory ICs are still also connected to the GCL so they can continue to work as separate memory modules when not engaged in GCL $direct$ activity.

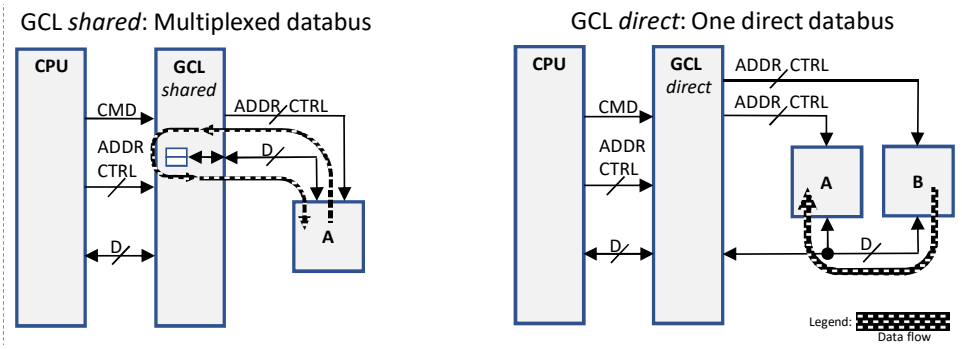

Figure 2: GCL $shared$ with $fromspace$ and $tospace$ sharing one memory module and GCL $direct$ with a bridged databus between each memory module, A and B, serving $fromspace$ and $tospace$ respectively

We evaluate the new $direct$ architecture against the $shared$ baseline architecture. The baseline architecture is when the one memory module hosts both of the semispaces, but has to do its job with one shared address- and databus. The GCL *direct* is where each memory module still has its own address and control path, but the data path is now connected to botyh memories. We compare (see Section 5) these solutions with respect to bus utilization, pin usage, logic element (LE) usage, and maximum achievable frequency.

4.2 Performance Analysis

The direct copying performance P in bytes/s (B/s) depends on the width W in bytes of the databus, the clock frequency f_{clk}, and the bus utilization U_{bus}. For example, if there are 25 wasted (e.g. overhead) clock cycles for each 100 cycles in a copy operation U_{bus} would be 75%.

$$P = W * f_{clk} * U_{bus} \tag{1}$$

For an example of a 36-bit SRAM with an actual datawidth of 4 bytes (B) since the 4 extra bits (almost always used for parity bits) are often *overhead* in terms of the object data word size, a clock speed of 100 MHz, and 75% efficiency of the copying (bus utilization), the copying performance, $P(MB/s)$, is

$$P = 4B * 100MHz * 0.75 = 300MB/s. \tag{2}$$

In the simulation analysis (see Section 5) W is set to 4 bytes, even if there are 4 extra parity bits available, f_{clk} is set to 200 MHz, and U_{bus} will vary depend on the design. This is also further quantified in Section 6 using benchmark data.

5 Garbage Collection Logic Simulation

We simulate the different architectures using SystemVerilog (IEEE, 2012) HDL and CAD tools.

5.1 Size and Data Width

SRAMs come in sizes ranging from smaller 2-Mb (megabit) units up to currently 144-Mb. Datawidth (often the same as word size) ranges from 18-bit to 72-bit. There is an insignificant reduction in the number of address pins needed for a 72-bit wide SRAM compared to a 18-bit word version, but it all adds up to less board space needed in an implementation. A promising property from a perspective of further reducing maximum GC blocking time is the idea of using 72-bit data width.

The other advantage of 18-bit, 36-bit, and 72-bit, vs. traditional 16-bit, 32-bit, and 64-bit memory widths is that these extra (parity) bits can be useful in aiding the GCL module during the GC process. Those extra 4 bits (if one is using a 36-bit memory module) can be used to *mark* or *tag* during the GC process: (1) They can tag which words hold a reference and (2) help identifying the references from the root set easier as opposed to manually walking the stack as in Pedersen and Schoeberl (2006). Some bits can change meaning depending on the state of the system such that even more use cases can be realized. To name one use case outside GC processing, one could use these bits to mark code and data coverage in response to test cases of code coverage and data flow analysis.

Least pins: The GCL $shared$ has one address bus and one data bus. The data is read from memory A into the GCL, and then (via a small FIFO) written back to $tospace$ also in the same A memory. The HW “cost" is 20 address lines, 36 data lines, and 8 control lines

(cf. Table 2) = **64 pins/balls**. However, the bus utilization in terms of copying is at most 50% ($U_{bus} = 0.50$ in Eq. 1).

Most pins: The GCL $direct$ has one shared bus line that is controlled concurrently by two independent address- and control-lines for A and B. The HW pin *cost* is
2 x 20 + 36 + 2 x 8 = **92 pins/balls**. However, the bus utilization in terms of copying is now at most 100% ($U_{bus} = 1.0$ in Eq. 1).

In Section 6 on experimental data the two upper bounds are reduced slightly due to extra cycles spent looking for live objects.

5.2 Memory Types and Signals

There are several different kinds of SRAM. Some are pipelined, some have zero-turn-around (ZBT) time between consecutive back-to-back read and write operations, and some have dual data ports (one for write and one for read). We will briefly outline the key functionality of these different SRAM types, and then discuss how these are implemented by one specific vendor, Integrated Silicon Solution Inc. (ISSI). The main signals and how they differ are discussed for the ISSI SRAMs. Similar SRAMs are offered by other market leading vendors, such as Cypress.

The pipelined SRAMs are faster than the flow-through SRAMs. A pipelined SRAM can operate at 200 (or 266) MHz which enables (even) faster GC than if a flow-through SRAM of 100 or 133 MHz is used. The ZBT no-wait option is useful for fast switching between read and write cycles that are interleaved, as needed for the GCLs in Figure 2, but also during normal mutator operation with constant load/store operations to the stack(s) and heap(s). This comes at the expense of some more complexity in the implementation. If we have a setup with a shared data bus and shared address bus, then we are better off with the ZBT no-wait options for both the flow-through and the pipelined SRAMs.

Neither GC $shared$ nor GC $direct$ has to make use of the chip select lines CE, $CE2$, and $CE2_n$. GC $shared$ does not use them because there is only one memory IC in that particular canonical setup. GC $direct$ does not need to disable either of the two memory ICs. Actually, when GC $direct$ copies live objects between the two semispaces, it needs to

Table 2 Main signals for memory control

Symbol	Note
A	Address bus
$\overline{CKE}$	$HIGH$: no changes in clock enable (keeping its state)
ADV	$HIGH$: burst counter increments. New addresses are loaded when ADV is low
	LOW: new addresses are loaded
$\uparrow \overline{WE}$	LOW: data is written into the system (all $\overline{BW}$ are low)
$\overline{OE}$	LOW: asynchronous output is enabled
DQ	Data inputs/outputs
BW_{a-d}	Byte write
$\uparrow \overline{CE}$	Not chip enable.

have both memory ICs enabled. One the other hand, it would be possible that for example memory IC B was disabled when memory IC A was used for normal activities by the mutator. This would further reduce power consumption when no GC process is active. But this is not necessary because it is easy to “NOP" either of the memory chips by driving $\overline{OE}$ high while performing a dummy read operation. As the chosen IC then tri-states its outputs, the other memory IC(s) can freely use the databus for other purposes than $direct$ GC activity. All in all, this can save another 2 x 3 = 6 control pins in the GC $direct$ setup. One pin can be saved and that is the clock enable pin $\overline{CKE}$, since the clocking of the memory ICs is on at all times during either meaningful operations or NOP operations.

6 GC Trace Benchmark

This section is dedicated to the various experiments on the GCL $shared$ and $direct$ architectures. To create a realistic test scenario, we need experimental data which can give us information about GCL $shared$ and GCL $direct$ performance in the managed language scenario.

To this end, we use data from two papers (Li et al., 2013; Sarimbekov et al., 2013) which use the *Elephant Track* software (Ricci et al., 2011). The work by Li et al and Sarimbekov et al describes the object lifetime of JVM objects in terms of known benchmark suites. When we combine information about *typical* size (i.e., between first and third quartile of box plots) and the *nursery* information, we get valuable information on how to test the proposed GCL architectures. Blackburn et al. (2004) describe the nursery as follows: “The generational collectors divide newly allocated *nursery* objects from mature objects that survive one or more collections, and collect the nursery independently and more frequently than the mature space".

Li et al. configure the nursery size to 4 MB. We do the same for the SRAM that we are modelling as it allows us to use the collected numbers per benchmark for the nursery collection. Sarimbekov et al. collected object sizes for the same benchmarks and we use the typical interval of sizes in their study to generate simulation data. We fill a heap with 4 MB of objects generated according to the studies of how Java, Closure, Jython, and JRuby compare across the benchmarks of spectralnorm, revcomp, regexdna, nbody, knucleotide, fasta, and binarytrees. It is impossible for us to know which benchmark might be more likely in a real world situation, and accordingly we sample uniformly and generate objects from each of these 7 benchmarks with equal probability. We also only included benchmarks which were reported in *both* studies *and* for all 4 languages (Java, Closure, Jython, and JRuby). Listing 1 shows how we generated test data for the GCL simulation.

Listing 1: Object table generation for Closure simulation

```
1  //CLOSURE benchmark heap generation
2  //spectralnorm,revcomp,regexdna,nbody,knucleotide,fasta,binarytrees
3  parameter NUSERY_CUT //Nursery: see Sarimbekov et al 2013 reference
4    =(0.1007+0.1385+0.3365+0.0339+0.0628+0.0450+0.0272)/7.0;
5  integer unsigned objsize[][2] //Obj. sizes: see Li et al 2013 reference
6    ='{'{24,48},'{24,48},'{24,48},'{24,24},'{24,24},'{24,32},'{24,24}};
7  //... some declarations omitted ...
8  task initObjHandles();
9    adroffset=0;
10   for (int unsigned index=0; index<MAXNUMOBJ; index++) begin
11     bench= //sample random object from any benchmark
12       $urandom_range($size(objsize,1)-1,0);
13     benchsize= // sample a size within boxed sizes
14       objsize[bench][$urandom_range($size(objsize[bench],1)-1,0)]/4;
15     if ($random<NUSERY_CUT)
16       mark=1; //will survive collection (->tospace)
17     else
18       mark=0;
19     objref[index]<='{oid:index,adr:adroffset,size:benchsize,mrk:mark};
20     adroffset=adroffset+benchsize;
21   end
22 endtask
```

Table 3 shows the SystemVerilog simulation results with the test data from the benchmarks for the two architectures. We can observe that copying with GCL $direct$ takes about half the time of copying with GCL $shared$. This shows that we achieve our goal of a two-fold speedup. The number of allocated objects ranges from 124,436 to 141,110 for the benchmark data related to GCL $shared$. For GCL $direct$ the number of allocated objects ranges from 124,276 to 141,079. The GC collection time lies between 5.864 and 5.940 ms for GCL $shared$, and between 2.930 and 2.967 ms for the GCL $direct$ setup. This confirms the previous statement that the performance of GCL $direct$ is twice that of GCL $shared$. The test setup is broad in the sense that 4 different languages are included: Java, Closure,

Jython, and JRuby. Furthermore, we generate nursery heaps for these languages both for the GCL $shared$ and GCl $direct$ architectures.

Table 3 Experimental results for using GCL $shared$ and $direct$ on benchmark data

	Java[1]		Closure[1]		Jython[1]		JRuby[1]	
GCL	#obj.	time (ms)	#obj.	time (ms)	#obj.	time (ms)	#obj.	time (ms)
$shared$[2]	124,436	5.864	141,110	5.940	128,834	5.888	126,498	5.878
$direct$[2]	124,276	2.930	141,079	2.967	128,782	2.938	126,283	2.935

[1] See Li et al and Sarimbekov et al Li et al. (2013); Sarimbekov et al. (2013)
[2] Note that the numbers for *direct* copying (i.e. #obj : See Listing 1 and Section 9.

7 GCL Within Host FPGA

Using an FPGA as the host for the GCL module is a flexible solution. Assuming that one chooses an FPGA with a sufficient number of pins/balls, it is possible to have each memory module independently connected to the GCL FPGA and the copying process can consist of moving in data from one memory module and, on the next cycle, writing it out to the other memory module. However, one still has to decide on the command interface to the GCL FPGA. Should it be seen as a pure SRAM from the main CPU/FPGA or would one want to expose some control registers using an embedded protocol or bus standard as a command interface? We implement a solution that exposes the SRAMs with their standard signals such as write enable (WE), output enable (OE), etc., but we also provide extra control signals as needed for the GCL *direct* solution to work (see Figure 2).

Figure 3 show the main functionality of the different parts of the PCB. Of special interest is the direct GC connection. In addition the two SRAM modules from ISSI are visible adjacent to the Max10 FPGA from Intel.

7.1 GC Copying PCB Module

We implement the architecture(s) on the IS61NVP51236B-200B3 version of ISSI synchronous pipelined single cycle deselect SRAM. We chose the 2.5 V version over a 3.3 V

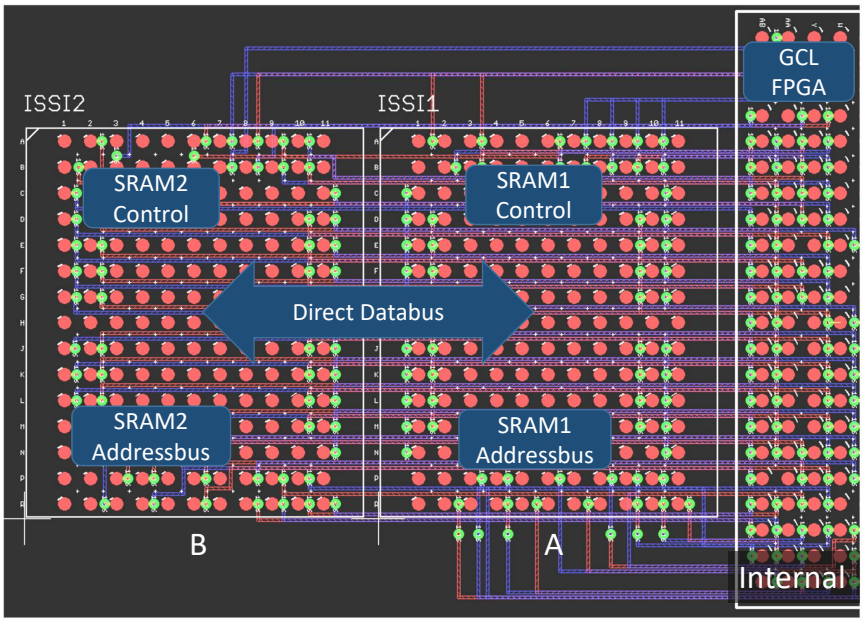

Figure 3: Overview of main functionality on GCL PCB

version to avoid having an extra voltage level present in the design. It was not possible to buy this memory IC in small quantities, however ISSI provided samples upon request.

PCB routing: In a CAD system, such as EagleCAD, the maximum number of layers for the Maker edition are 6, and to achieve routing from a high-density FPGA, the vias must be smaller. The routing is possible with a BGA with 1.0 mm pitch and 0.1 mm trace and trace clearance width.

Maximum clock frequency: It is possible to estimate the maximum clock speed for a design using the TimeQuest Timing Analyzer from the Altera/Intel Quartus Prime development suite (Altera, 2010). We have performed this static timing analysis for the design called GCL *direct* (see Section 4) and the maximum frequency is estimated to be between 296 MHz to 330 MHz, depending on the operating temperature.

Resource usage: The implemented GCL *direct* module uses 3 input pins, 56 output pins, 36 bidirectional pins, and 156 logic elements. The Max10 devices have between 2 K and 50 K logic elements.

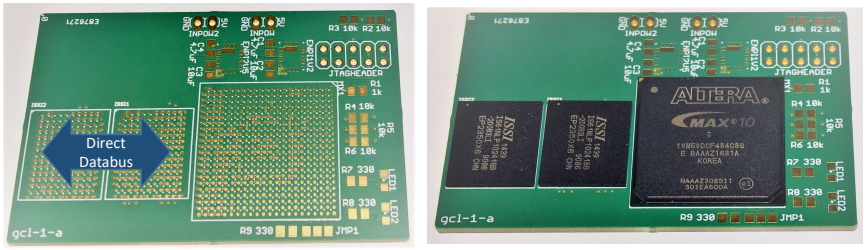

Figure 4: Printed circuit boards for GCL *direct*

8 Discussion

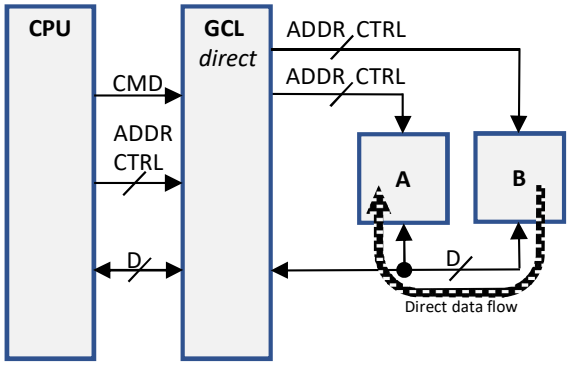

Figure 5:

The chosen approach was to move a complete *nursery* region (i.e. all the live-objects) from memory A to memory B (see Section 3). In this case we found that 4 MB was a valid size based on the numbers from the *Elephant track* software (Li et al. (2013); Sarimbekov et al. (2013)). Each of the copied objects are handled according to the process illustrated in Figure 1.

As we evaluated the two solutions relative to one other (see Figure 2), based on the resources used (e.g. board space and LEs), speed, and implementation complexity, we found that *GCL direct* was closer to optimal. This is summarized in Table 4.

In Section 5, we considered the influence of width (the number of bits and size of the databus) as a design parameter. The fundamental speed of the copying process is of course

Table 4 Comparison between GC $shared$ and the new GC $direct$ architectures on speed and complexity

Solution	GCL *shared*	GCL *direct*
Speed[1]	◐	● [2]
Complexity[3]	◔	◐

[1] Measured in bytes per second.
[2] More black is better.
[3] Measured as pins used.

directly proportional to the width of the databus. If size permits, it would be advantageous to go for SRAMs with 72-bit widths over both 18-bit and 36-bit.

The actual two-fold speedup is documented both in the analysis section, Section 4.2, and by the GC trace benchmark simulation in Section 6. The simulation confirmed the analysis that (a) GC $direct$ was twice as fast as GC $shared$ and (b) the time used for the GC process is less than 3 ms even with a simulated size of the $nursery$ region of 4 MB, which is relatively large for embedded systems.

A realization of GC $direct$ on a recent FPGA from Intel (formerly Altera) was possible on a 6-layer board offering evidence that GC $direct$ is also practical. Normally, such designs would use up to 8-16-layers, but the open source hardware files we provide show how to route and fabricate the real PCB using only the Maker edition of EagleCAD.

8.1 Future Work

Future work in terms of functionality for the embedded systems discipline should include extending the GCL $direct$ architectures to other kinds of memory such as different DDR. Some DDR memories now have an SRAM-like interface making them easier to work with compared to previous DDR memories. Furthermore, they are larger (over 1 Gb) compared to the available SRAMs (72-Mb to 144-Mb). This can be useful in certain multicore systems (e.g. *Patmos* by Schoeberl et al. (2015)) or certain datastreaming systems, such as new embedded systems similar to $DEMoS$ (Ulslev Pedersen, 2016), and also in hard real time data mining systems Pedersen (2006).

9 Conclusion

As the Internet of Things (IoT) and cyber-physical systems are on the rise (Fiedler and Meissner, 2013; Latronico et al., 2015), so is the need for better and more efficient software and hardware in this space. This development can be supported by managed languages that offer advantages such as object orientation and managed memory systems with garbage collection (GC). In this paper we have worked with, and provided an architecture for, SRAM ICs to provide solutions to an important problem: That the SRAM is a major bottleneck in managed language GC-based systems. GC is a perpetual bottleneck, and we have analyzed, simulated, and implemented a new architecture, **GC** $direct$**, that offers a two-fold speedup**, with no significant drawbacks in complexity except for the pins needed for an extra set of extra address- and control-lines.

The two conclusive things to be learnt from this paper in terms of designing for minimizing the maximum GC time can be summarized in two design strategies for configuring SRAMs in order to achieve fast GC:

① Consider using zero-wait-state SRAM with the GC $direct$ setup to achieve an important two-fold speedup as compared to GC $shared$.

② Consider using an FPGA to implement the GCL $direct$ module.

Upon publication, the open source code for this paper is available for download at Github `https://github.com/gclrt/gcl1` and it can be tested at EDA Playground `https://www.edaplayground.com/x/uEh`.

The suite of CAD tools used in this paper were ModelSim Intel FPGA Starter edition 10.5b, Quartus Prime 16.1.1, Aldec Riviera Pro 2015.06 on EDA playground, and Autodesk Eagle 8.0.0. Eurocircuits produced the PCBs. The FPGA is an Max 10 10M50 with 484 balls/pins.

References

Altera. Guaranteeing Silicon Performance with FPGA Timing Models. Technical report, 2010. URL https://www.altera.com/en{_}US/pdfs/literature/wp/wp-01139-timing-model.pdf.

D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-628-5. doi: http://doi.acm.org/10.1145/604131.604155.

D. F. Bacon, P. Cheng, and S. Shukla. And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 23–34, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254068.

H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4): 280–294, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359460.359470.

S. M. Blackburn, P. Cheng, and K. S. Mckinley. Myths and Realities : The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '04/Performance '04*, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005693.

M. Fiedler and S. Meissner. IoT in Practice: Examples: IoT in Logistics and Health. In *Enabling Things to Talk*, pages 27–36. 2013. ISBN 978-3-642-40402-3. doi: 10.1007/978-3-642-40403-0_4. URL http://link.springer.com/10.1007/978-3-642-40403-0.

A. Goldberg and D. Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley, 1983. ISBN 0201113716. URL http://www.mirandabanda.org/bluebook/.

T. Gomes, J. Pereira, P. Garcia, F. Salgado, V. Silva, S. Pinto, M. Ekpanyapong, and A. Tavares. Hybrid real-time operating systems: deployment of critical FreeRTOS features on FPGA. *International Journal of Embedded Systems*, 8(5/6):483, 2016. ISSN 1741-1068. doi: 10.1504/IJES.2016.080386.

M. T. Hoske. Industry 4.0 and Internet of Things tools help streamline factory automation. *Control Engineering*, 62(2):M7–M10, 2015. ISSN 00108049. doi: 10.1007/978-3-319-42559-7.

L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7), pages 73–82, San Diego, CA, May 1993.

IEEE. IEEE SA - 1800-2012 - IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language, 2012. URL `https://standards.ieee.org/findstds/standard/1800-2012.html`.

R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. ISBN 0–471–94148–4. With a chapter on Distributed Garbage Collection by R. Lins.

E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A Vision of Swarmlets. *IEEE Internet Computing, Special Issue on Building Internet of Things Software*, 19(2):20–29, mar 2015. URL `http://terraswarm.org/pubs/332.html`.

E. A. Lee. Cyber Physical Systems: Design Challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008. ISBN 9780769531328. doi: 10.1109/ISORC.2008.25. URL `http://chess.eecs.berkeley.edu/pubs/427.html`.

W. H. Li, D. R. White, and J. Singer. JVM-hosted languages. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java*

Platform Virtual Machines, Languages, and Tools - PPPJ '13, pages 101–112, New York, New York, USA, 2013. ACM Press. ISBN 9781450321112. doi: 10.1145/2500828.2500838. URL http://dl.acm.org/citation.cfm?doid=2500828.2500838.

M. Maas, K. Asanovic, and J. Kubiatowicz. Grail quest: A new proposal for hardware-assisted garbage collection. In *6th Workshop on Architectures and Systems for Big Data (ASBD '16)*, Seoul, Korea, June 2016.

M. Meyer. A true hardware read barrier. In E. Petrank and J. E. B. Moss, editors, *Proceedings of the 5th International Symposium on Memory Management (ISMM 2006)*, pages 3–16. ACM, June 2006. ISBN 1-59593-221-6.

K. D. Nilsen and W. J. Schmidt. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*, 1(4):338–354, Dec. 1992.

R. Pedersen and M. Schoeberl. Exact roots for a real-time garbage collector. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 77–84, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-544-4. doi: 10.1145/1167999.1168013.

R. Pedersen, J. Nørbjerg, and M. Scholz. Embedded programming education with Lego Mindstorms NXT using Java (leJOS), eclipse (XPairtise), and python (PyMite). In *Proceedings - 2009 Workshop on Embedded Systems Education, WESE 2009*, 2009. ISBN 9781450300216. doi: 10.1145/1719010.1719019.

R. U. Pedersen. Hard real-time analysis of two java-based kernels for stream mining. In *Proceedings of the 1st Workshop on Knowledge Discovery from Data Streams (IWKDDS, ECML PKDD 2006)*, Berlin, Germany, September 2006.

R. U. Pedersen. Micro information systems and ubiquitous knowledge discovery. In M. May and L. Saitta, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6202 LNAI,

pages 216–234. Springer, 2010. ISBN 3642163912. doi: 10.1007/978-3-642-16392-0_13.

N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java - PPPJ '11*, page 139, New York, New York, USA, 2011. ACM Press. ISBN 9781450309356. doi: 10.1145/2093157.2093178. URL http://dl.acm.org/citation.cfm?doid=2093157.2093178.

A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder. Characteristics of dynamic JVM languages. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages - VMIL '13*, pages 11–20, New York, New York, USA, 2013. ACM Press. ISBN 9781450326018. doi: 10.1145/2542142.2542144. URL http://dl.acm.org/citation.cfm?doid=2542142.2542144.

M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008. doi: http://dx.doi.org/10.1016/j.sysarc.2007.06.001.

M. Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3): 176–213, 2010. doi: 10.1007/s11241-010-9095-4.

M. Schoeberl and W. Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, pages 77–84, Santa Clara, California, September 2008. ACM Press. doi: 10.1145/1434790.1434802.

M. Schoeberl and W. Puffitsch. Nonblocking real-time garbage collection. *ACM Trans. Embed. Comput. Syst.*, 10(1):6:1–28, 2010. ISSN 1539-9087. doi: 10.1145/1814539.1814545.

M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki,

J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. ISSN 1383-7621. doi: http://dx.doi.org/10.1016/j.sysarc.2015.04.002.

F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems (CASES 2000)*, pages 9–17, New York, NY, USA, 2000. ACM. ISBN 1-58113-338-3. doi: http://doi.acm.org/10.1145/354880.354883.

F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books, 2002.

SPEC. The spec jvm98 benchmark suite. Available at https://www.spec.org/jvm98/, August 1998.

M. Swan. Sensor Mania! The Internet of Things, Wearable Computing, Objective Metrics, and the Quantified Self 2.0. *Journal of Sensor and Actuator Networks*, 1(3):217–253, 2012. ISSN 2224-2708. doi: 10.3390/jsan1030217. URL `http://www.mdpi.com/2224-2708/1/3/217/htm`.

R. Ulslev Pedersen. DEMoS Manifesto. *ArXiv e-prints*, dec 2016. URL `https://arxiv.org/abs/1612.04191`.

P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, Jan. 1994. Expanded version of the IWMM92 paper.

T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990. doi: 10.1016/0164-1212(90)90084-Y.