Direct garbage collection: two-fold speedup for managed language embedded systems

Rasmus Ulslev Pedersen*

Department of Digitalization,
Copenhagen Business School, Denmark
Email: rup.itm@cbs.dk
*Corresponding author

Martin Schoeberl

Department of Applied Mathematics and Computer Science,
Technical University of Denmark, Denmark
Email: masca@dtu.dk

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 databusses. This direct-path memory setup is generalisable, 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 realisation of the direct fast copy architecture.

Keywords: garbage collection; managed languages; embedded systems; real-time; memory; SRAM.

Reference to this paper should be made as follows: Pedersen, R.U. and Schoeberl, M. (2018) 'Direct garbage collection: two-fold speedup for managed language embedded systems', *Int. J. Embedded Systems*, Vol. 10, No. 5, pp.394–405.

Biographical notes: Rasmus Ulslev Pedersen works at Copenhagen Business School since 2005 to present. He has given PhD level summer school classes on intelligent systems. At Department of Digitalisation, he has taught IT, business intelligence, internet of things, and data mining. He is a co-founder of the Internet of Things Forum and IoT People. He has served on several EU FP7 projects (i.e., kdUBIQ and FINODEX).

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 as an 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. He 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, which 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 miniaturisation 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 realise 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.

3 A prototype implementation on real hardware based on our open source software/hardware.

The paper is organised 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 realisation 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 7. 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

GC research started with collectors for LISP and ML (Moon, 1984). 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 minimise blocking times due to the GC activity. A good introduction to GC 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 (1992) 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 so-called nursery regions, 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 utilisation (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.

As in Meyer's proposal, Maas et al. (2016) propose to add hardware support for GC *into* a standard processor. 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 soft core processors like JOP or Altera/Intel Nios II.

It is also possible that direct memory access (DMA) can prove useful for the GC copy operation for standard CPUs as well as FPGAs. DMAis generally used to speed up memory copy and move operations. For instance, it is discussed as a hardware unit within a processor by Su et al. (2011) and Li et al. (2017). Recent work by Li et al. (2017) also suggests chip multi-processor (CMP) support via message passing mechanisms. This aspect (provision of a message passing method between GC modules) is outside the scope of this paper. It is worth nothing though, that the message passing paradigm also relies on RAM for extra functionality, similarly to what we present later in this paper.

Bacon et al. 2012 present a GC implemented in an FPGA. 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 GC: 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.

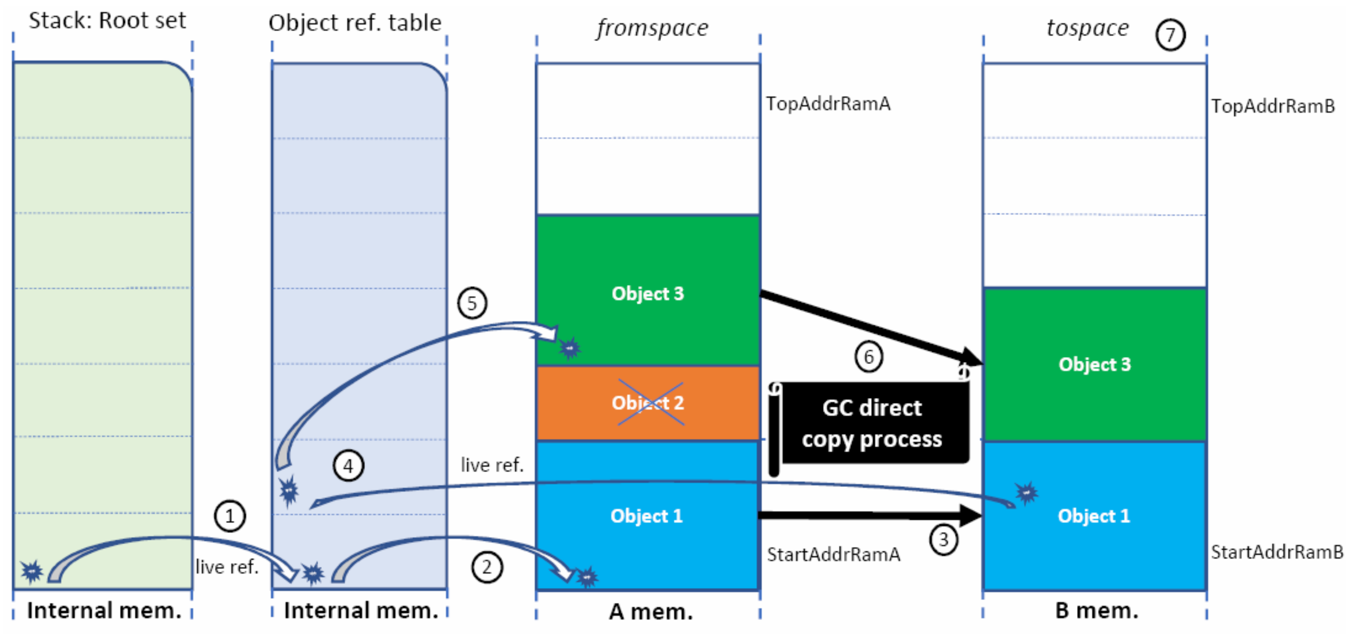
In the figure, three objects are hosted in the *fromspace*. Objects 1 and 3 are alive (a *live* reference points to each) and can be accessed from a running program (i.e., a mutator thread). During GC, These two objects will be copied to semispace *tospace*. Then the memory area of semispace *fromspace*, 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

b we speed up the copy process by a direct link between the two memory chips.

Figure 1 GC process (see online version for colours)



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.

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 optimise 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 nine 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 Table 1 that only four of the nine objects are alive, and that objects 0, 2, 4, 6, and 8 should be garbage collected.

After GC, i.e., copying the *live* objects to the other semispace, the memory will look as in Table 1 (right side). The four copied objects are all of different sizes and if the copy process were executed on a word-wide databus it would take one cycle/word to copy each object from *fromspace* to *tospace*.

Table 1 Copy objects marked alive from *fromspace* to *tospace*

Fromspace					*Tospace*			
Obj. ID	*Addr*	*Size*	*Alive*		*Obj. ID*	*Addr*	*Size*	*Alive*
0	0 × 000	0 × 10	0		1	0 × 000	0 × 20	1
1	0 × 010	0 × 20	1		3	0 × 020	0 × 40	1
2	0 × 030	0 × 30	0		5	0 × 060	0 × 60	1
3	0 × 060	0 × 40	1	⇒	7	0 × 0C0	0 × 80	1
4	0 × 0A0	0 × 50	0					
5	0 × 0F0	0 × 60	1					
6	0 × 150	0 × 70	0					
7	0 × 1C0	0 × 80	1					
8	0 × 240	0 × 90	0					

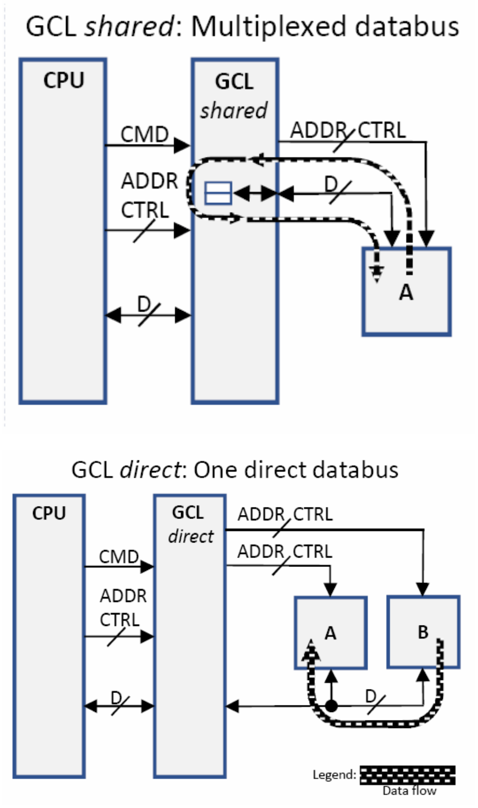
4 GC copy performance analysis

This section discusses aspects of the design space w.r.t. GC performance. Our quantification includes databus width, operating frequency, and possible bus utilisation. Figure 2 shows two different architectures for the GCL:

1 the GCL *shared*

2 the GCL *direct* architectures.

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 (see online version for colours)



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 **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 memories 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_{DQ1} to B_{DQ1} 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.

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 both memories. We compare (see Section 5) these solutions with respect to bus utilisation, 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 utilisation 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 four 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 utilisation), the copying performance, P (MB/s), is

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

In the simulation analysis (see Section 5) W is set to 4 bytes, even if there are four 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 GCL simulation

We simulate the different architectures using SystemVerilog (IEEE, 2012) HDL and CAD tools. In particular, we use the ModelSim Intel FPGA Starter edition 10.5b as well as Aldec Riviera Pro 2015.06 on EDA playground for the simulation work.

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 an 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.

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 realised. 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.

- *Fewest 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 utilisation in terms of copying is at most 50% [U_{bus} = 0:50 in equation (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 $2 \times 20 + 36 + 2 \times 8 =$ **92 pins/balls**. However, the bus utilisation in terms of copying is now at most 100% [U_{bus} = 1.0 in equation (1)].

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

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.

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*, *CE*2, 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 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 $2 \times 3 = 6$ 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. (2013) and Sarimbekov et al. (2013) 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. (2013) 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. (2013) 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 seven benchmarks with equal probability. We also only included benchmarks which were reported in *both* studies *and* for all four 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

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

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

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

Notes: [2]Note that the numbers for direct copying [i.e., # obj and time (ms)] for Java, Closure, Jython and JRuby are not exactly equal because they are generated probabilistically: see Listing 1 and Section 9.

Source: [1]See, Li et al. (2013) and Sarimbekov et al. (2013)

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 four 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.

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 Overview of main functionality on GCL PCB (see online version for colours)

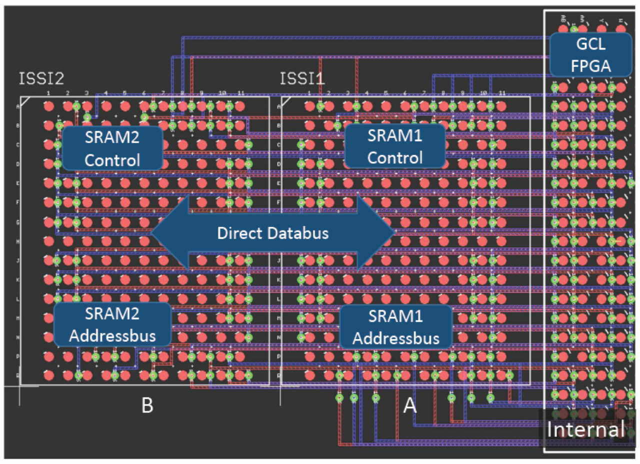


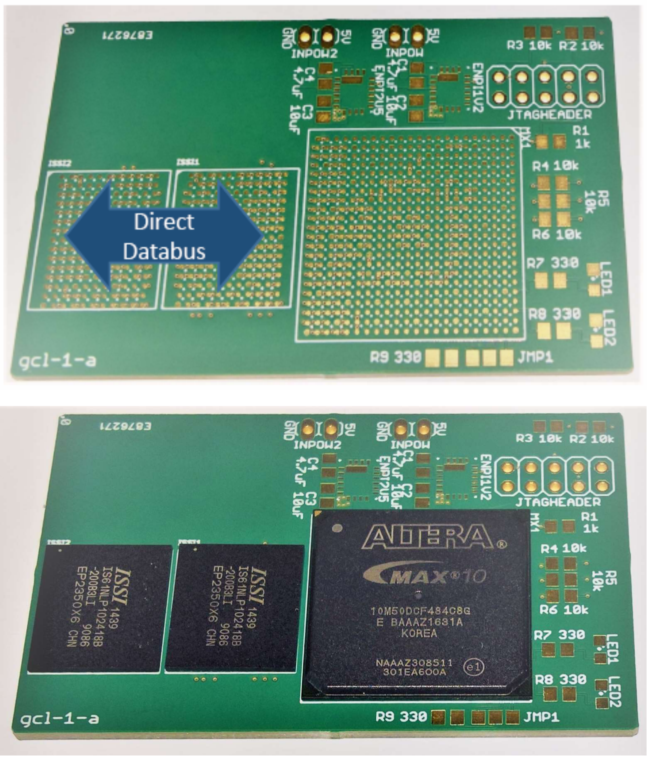
Figure 3 show the main functionality of the different parts of the PCB. Of special interest is the direct GC connection. In addition the two SRAM modules from ISSI are visible adjacent to the Max10 FPGA from Intel. For this section, Autodesk EagleCAD 8.0.0 is used to create the PCB schematic and board files, while Eurocircuits produces the PCB from the Gerber files.

7.1 GC copying PCB module

We implement the architecture(s) on the IS61NVP51236B-200B3 version of ISSI synchronous pipelined single cycle deselects SRAM. We chose the 2.5Vversion over a 3.3V 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 six, 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 Analyser 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 296MHz to 330MHz, 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 2K and 50K logic elements. The specific FPGA used in this paper is the Max10-10M50 with 484 balls/pins.

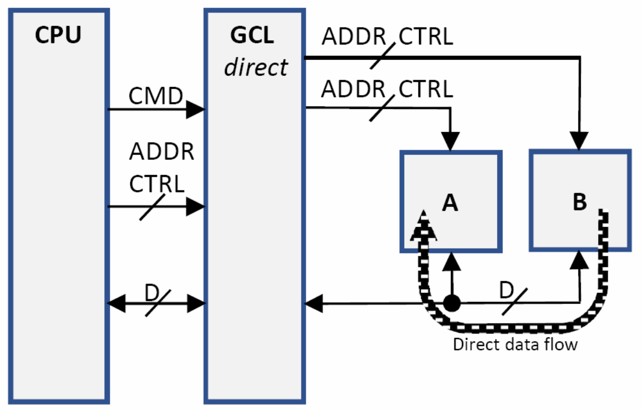
Figure 4 Printed circuit boards for GCL *direct* (see online version for colours)



8 Discussion

It is generally recognised that hardware can often speed up critical sections of an embedded system relative to a software implementation. This was discussed in Section 1 as a way to achieve the *two-fold speedup for managed language embedded systems*. Gomes et al. (2016) note that "Performance and determinism are two critical metrics in most embedded systems with real-time requirements". We address the performance issue by a reduction in latency that is due to the parallel *autonomy* of the *direct* GCL module (see Figure 5). Another aspect of GC is the issue of *fragmentation* (see Section 2) that we address by implementing a stop-the-world type GC. A blocking time in the region of 3 ms was achieved. This is a low value (see Table 3), and it must be noted that it could be even lower, depending on the design choices. 'Activating' two design choices of doubling the system frequency to 200 MHz and doubling the datawidth to the SRAM *would result in a blocking time well under 1 ms*. However, the 'price' to be paid would be higher complexity, as the SRAMs would then have to be pipelined. This would not be a potential issue for stack operations if the same SRAM IC were also used for stack *fill* and *spill* operations as this particular choice could increase min. latency in other parts of the embedded system. Still, it is attractive in certain scenarios to be able to design for a full GC cycle that lasts less than 1 ms.

Figure 5 GCL direct: the main designed artefact (see online version for colours)



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 is 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. Of the parameters which were compared, GC copy speed, as a performance metric, *P*, measured in *B/s*, is the most important direct parameter. We found that this parameter was 300 MB/s within our setup. The price (in hardware) is that we use 16 more balls for the GCL *direct* than we do for the GCL *shared* solution, and this extra use of I/O balls is because the GCL *shared* only has one set of shared address and control pins (there is only one memory IC). It should be noted that there is one more design parameter which could be invoked: *multiplexing* of the address and control lines to memories A and B. This would reduce the I/O ball 'cost' for GCL *direct* to be comparable with GCL *shared*. A potential drawback of such a design is higher complexity and the enforced use of the SRAM *bursting* functionality (see IS61NVP51236B-200B3 in Section 7.1), which is when a memory IC is provided with a start address for either a read or write operation and then automatically increments the address counter as it sequentially reads (or writes) four words. We have explored this possibility, but the need for a high number (one for each address pin and control pin) of extra SMD flip-flops and tri-state buffers on the PCB made this approach inferior in relation to the central message of this paper: the double speedup achieved from a simple GCL *direct* architecture. This is summarised in Table 4.

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]	◔	◐

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

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 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*

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 realisation of GC *direct* on a recent FPGA from Intel (formerly Altera) was possible on a six-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. In retrospect, we would not start out with a complex design on a six-layer PCB, as the routing was challenging (two address/control buses and one data bus means a lot of connections going in one direction).We would recommend to use at least an eight-layer board as the starting point. Finally, as noted earlier, we used 2.5 V versions of the ISSI SRAM. This is not a problem in small scale designs, but it would be easier to use 3.3 V ICs if one prioritised these ICs being available on large electronics marketplaces such as

Digikey. The design was fast and static timing analysis placed it around 300 MHz, and in that sense it would be possible to use even the fastest SRAMs which are capable of running at 266 MHz. This would bring the added benefit of not having to use multiple clock domains (i.e., the GCL module would run at the same high frequency as the SRAM ICs).

8.1 Future work

One can divide future work into functionality and physical hardware issues. Functionality issues include support for more complicated and complex memory types such as the DDRtypes. Hardware aspects include design support for placing GC-modules dynamically on chip at runtime.

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 multi-core systems [e.g., *Patmos* by Schoeberl et al. (2015)] or certain data streaming systems, such as new embedded systems similar to *DEMoS* (Pedersen, 2016), and also in hard real-time data mining systems (Pedersen, 2006).

On systems with long deployment times one cannot always know how many cores are needed and how many of those cores will need a dedicated GC-module. One way to address this issue is to 'reserve' space on the FPGA logic area for such GC-modules. It could possibly be achieved with so-called *forbidden zones* as presented by Ouni and Mtibaa (2015). It is possible this approach will allow the GC-modules to be placed at reserved spots close to the pin-outs of the dedicated memory ICs, such that the maximum frequency is still as high as possible (see 7.1).

9 Conclusions

As the 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 GC. In this paper we haveworked 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 analysed, 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 minimising the maximum GC time can be summarised 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.

All source code developed for this paper is available as open source (see Appendix).

References

Altera (2010) *Guaranteeing Silicon Performance with FPGA Timing Models*, Technical Report [online] https://www.altera.com/en{_}US/pdfs/literature/wp/wp-01139-timing-model.pdf (accessed August 2016).

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

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

Baker, H.G. (1978) 'List processing in real-time on a serial computer', *Commun. ACM*, Vol. 21, No. 4, pp.280–294, ISSN: 0001-0782, DOI: http://doi.acm.org/10.1145/359460.359470.

Blackburn, S.M., Cheng, P. and Mckinley, K.S. (2004) '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, ACM, New York, USA, pp.25–36, ISBN: 1-58113-873-3, DOI: 10.1145/1005686.1005693.

Fiedler, M. and Meissner, S. (2013) 'IoT in practice: examples: IoT in logistics and health', in *Enabling Things to Talk*, pp.27–36, ISBN: 978-3-642-40402-3, DOI: 10.1007/978-3-642-40403-0_4 [online] http://link.springer.com/10.1007/978-3-642-40403-0 (accessed August 2016).

Goldberg, A. and Robson, D. (1983) *Smalltalk-80: the Language and its implementation*, Addison-Wesley, ISBN 0201113716 [online] http://www.mirandabanda.org/bluebook/ (accessed August 2016).

Gomes, T., Pereira, J., Garcia, P., Salgado, F., Silva, V., Pinto, S., Ekpanyapong, M. and Tavares, A. (2016) 'Hybrid real-time operating systems: deployment of critical Free RTOS features on FPGA', *International Journal of Embedded Systems*, Vol. 8, Nos. 5/6, p.483, ISSN: 1741-1068, DOI: 10.1504/IJES.2016.080386.

Hoske, M.T. (2015) 'Industry 4.0 and internet of things tools help streamline factory automation', *Control Engineering*, Vol. 62, No. 2, pp.M7–M10, ISSN: 00108049, DOI: 10.1007/978-3-319-42559-7.

Huelsbergen, L. and Larus, J.R. (1993) 'A concurrent copying garbage collector for languages that distinguish (im)mutable data', in *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May, Vol. 28, No. 7, pp.73–82.

IEEE (2012) *IEEE SA – 1800-2012 – IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language* [online] https://standards.ieee.org/findstds/standard/1800-2012.html (accessed August 2016).

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

Latronico, E., Lee, E.A., Lohstroh, M., Shaver, C., Wasicek, A. and Weber, M. (2015) 'A vision of swarmlets', *IEEE Internet Computing, Special Issue on Building Internet of Things Software*, March, Vol. 19, No. 2, pp.20–29 [online] http://terraswarm.org/pubs/332.html (accessed December 2016).

Lee, E.A. (2008) 'Cyber physical systems: design challenges', in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp.363–369, ISBN: 9780769531328, DOI: 10.1109/ISORC.2008.25 [online] http://chess.eecs.berkeley.edu/pubs/427.html (accessed December 2016).

Li, W.H., White, D.R. and Singer, J. (2013) '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*, ACM Press, New York, USA, pp.101–112, ISBN: 9781450321112, DOI: 10.1145/2500828.2500838 [online] http://dl.acm.org/citation.cfm?doid=2500828.2500838 (accessed December 2016).

Li, Y., Zhang, Y., Jiang, C. and Zheng, W. (2017) 'Hardware support for message-passing in chip multi-processors', *International Journal of High Performance Computing and Networking*, Vol. 10, Nos. 4/5, p.391, ISSN: 1740-0562, DOI: 10.1504/IJHPCN.2017.086543.

Maas, M., Asanovic, K. and Kubiatowicz, J. (2016) '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.

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

Moon, D.A. (1984) 'Garbage collection in a large lisp system', in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, ACM, New York, USA, pp.235–246, ISBN: 0-89791-142-3, DOI: http://doi.acm.org/10.1145/800055. 802040 (accessed December 2016).

Nilsen, K.D. and Schmidt, W.J. (1992) 'Cost-effective object space management for hardware assisted real-time garbage collection', *ACM Letters on Programming Languages and Systems*, December, Vol. 1, No. 4, pp.338–354.

Ouni, B. and Mtibaa, A. (2015) 'Modules placement technique under constraint of FPGA forbidden zones', *International Journal of Computational Science and Engineering*, Vol. 11, No. 2, p.124, ISSN: 1742-7185, DOI: 10.1504/IJCSE.2015.071876.

Pedersen, R. and Schoeberl, M. (2006) '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)*, ACM Press, New York, USA, pp.77–84, ISBN 1-59593-544-4, DOI: 10.1145/1167999.1168013.

Pedersen, R., Nørbjerg, J. and Scholz, M. (2009) '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, ISBN: 9781450300216, DOI: 10.1145/1719010.1719019.

Pedersen, R.U. (2006) '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.

Pedersen, R.U. (2010) 'Micro information systems and ubiquitous knowledge discovery', in May, M. and Saitta, L. (Eds.): *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 6202 LNAI, Springer, pp.216–234, ISBN: 3642163912, DOI: 10.1007/978-3-642-16392-0_13.

Pedersen, R.U. (2016) *DEMoS Manifesto*, ArXiv e-prints, December [online] https://arxiv.org/abs/1612.04191 (accessed December 2016).

Ricci, N.P., Guyer, S.Z. and Moss, J.E.B. (2011) 'Elephant Tracks', in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java – PPPJ '11*, p.139, ACM Press, New York, USA, ISBN: 9781450309356, DOI: 10.1145/2093157.2093178 [online] http://dl.acm.org/citation.cfm?doid=2093157.2093178 (accessed December 2016).

Sarimbekov, A., Podzimek, A., Bulej, L., Zheng, Y., Ricci, N. and Binder, W. (2013) 'Characteristics of dynamic JVM languages', in *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages - VMIL '13*, pp.11–20, ACM Press, New York, USA, ISBN: 9781450326018, DOI: 10.1145/2542142.2542144 [online] http://dl.acm.org/citation.cfm?doid=2542142.2542144 (accessed December 2016).

Schoeberl, M. (2008) 'A Java processor architecture for embedded real-time systems', *Journal of Systems Architecture*, Vol. 54, Nos. 1–2, pp.265–286, DOI: http://dx.doi.org/10.1016/j.sysarc (accessed 6 January 2007).

Schoeberl, M. (2010) 'Scheduling of hard real-time garbage collection', *Real-Time Systems*, Vol. 45, No. 3, pp.176–213, DOI: 10.1007/s11241-010-9095-4.

Schoeberl, M. and Puffitsch, W. (2008) '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)*, ACM Press, Santa Clara, California, September 2008, pp.77–84, DOI: 10.1145/1434790.1434802.

Schoeberl, M. and Puffitsch, W. (2010) 'Non-blocking real-time garbage collection', *ACM Trans. Embed. Comput. Syst*., Vol. 10, No. 1, pp.6:1–28, ISSN: 1539-9087, DOI: 10.1145/ 1814539. 1814545.

Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., R. Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P., Rocha, Silva, A.C., Sparsø, J. and Tocchi, A. (2015) 'T-CREST: time-predictable multi-core architecture for embedded systems', *Journal of Systems Architecture*, Vol. 61, No. 9, pp.449–471, ISSN: 1383-7621, DOI: http://dx. doi.org/10.1016/j.sysarc.2015.04.002.

Siebert, F. (2000) '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)*, ACM, pp.9–17, New York, USA, ISBN: 1-58113-338-3, DOI: http://doi.acm.org/10.1145/354880.354883.

Siebert, F. (2002) *Hard Real-time Garbage Collection in Modern Object Oriented Programming Languages*, Aicas Books, SPEC, The spec jvm98 benchmark suite, ISBN: 3-8311-3893-1 [online] https://www.spec.org/jvm98/, (accessed August 2012).

Su, W., Wang, L., Su, M. and Liu, S. (2011) 'A processor-DMA-based memory copy hardware accelerator', in *Proceedings 6th IEEE International Conference on Networking, Architecture, and Storage*, NAS 2011, pp.225–229, ISBN: 9780769545097, DOI: 10.1109/NAS. 2011.15.

Swan, M. (2012) 'Sensor Mania! The internet of things, wearable computing, objective metrics, and the quantified self 2.0', *Journal of Sensor and Actuator Networks*, Vol. 1, No. 3, pp.217–253, ISSN: 2224-2708, DOI: 10.3390/jsan1030217 [online] http://www.mdpi.com/ 2224-2708/1/3/217/htm (accessed August 2012).

Wilson, P.R. (1994) *Uniprocessor Garbage Collection Techniques*, Technical Report, University of Texas, January, Expanded version of the IWMM92 paper.

Yuasa, T. (1990) 'Real-time garbage collection on general-purpose machines', *Journal of Systems and Software*, Vol. 11, No. 3, pp.181–198, DOI: 10.1016/0164-1212(90)90084-Y.

Appendix

Open source access

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 EDAplayground, and Autodesk Eagle 8.0.0. Eurocircuits produced the PCBs. The FPGA is an Max 10 10M50 with 484 balls/pins.