Designing a Concurrent Hardware Garbage Collector for Small Embedded Systems

Flavius Gruian and Zoran Salcic

Department of Electrical and Computer Engineering
The University of Auckland
Private Bag 92019, Auckland, New Zealand
{f.gruian, z.salcic}@auckland.ac.nz

Abstract. Today more and more functionality is packed into all kinds of embedded systems, making high-level languages, such as Java, increasingly attractive as implementation languages. However, certain aspects, essential to high-level languages are much harder to address in a low performance, small embedded system than on a desktop computer. One of these aspects is memory management with garbage collection. This paper describes the design process behind a concurrent, garbage collector unit (GCU), a coprocessor to the Java Optimised Processor. The GCU, targeting small embedded real-time applications, implements a mark-compact algorithm, extended with concurrency support, and tuned for improved performance.

1 Introduction

Java, as a development language and run-time solution, seems to become increasingly attractive recently, even for embedded systems, given the plethora of Java-powered embedded processors [1–4]. Nevertheless, few of these embedded platforms offer a true Java environment, including memory management with garbage collection. When present, garbage collection is in principle a software, stop-the-world approach, leading to poor performance systems. Although for high performance and desktop systems, both real-time and hardware supported garbage collection have been addressed by various research groups, there are few results for embedded systems with limited resources. In this paper, a concurrent garbage collection unit for the Java Optimised Processor (JOP, [5]) is described.

The paper is organised as follows. Section 2 mentions some of the relevant related work. The design methodology including goals and the design steps is given in Sect. 3. The used garbage collection algorithm is briefly described in Sect. 4, followed by the choices that remained unchanged throughout the design process in Sect. 5. The actual design iterations are detailed in Sect. 6. Section 7 discusses the implications of our solution, while Sect. 8 gathers our conclusions.

2 Related Work

Improving the performance of garbage collectors by using parallelism or concurrency came under the attention of researchers long before Java was born [6, 7].

In the context of garbage collection, we use *parallelism* to describe collection work done by several processors at the same time, while *concurrency* refers to running the application (mutator) at the same time with the collector.

Many of the concurrent GC algorithms have their roots in the famous *Baker's algorithm* [8], which is however unsuitable for embedded systems we are interested in, due to its high demands on the memory size. Non-copying concurrent garbage collection algorithms, with lower demands on the memory are described in [9] and [10]. We decided to implement an incremental version of a mark-compact algorithm (see [11]), which avoids the fragmentation issues that may arise in a non-copying algorithm.

Although hardware accelerated GC was used in early LISP and Smalltalk machines, one of the first to address it from a real-time perspective is [12]. That paper proposed a garbage collected memory module (GCMM), employing *Baker*'s semi-space algorithm. [13] proposes the Active Memory Processor, a hardware GC for real-time embedded devices. That approach uses hardware reference counting and a mark-sweep algorithm, requiring extra memory that scales with the heap. In contrast, our solution is independent of the heap size.

3 Design Methodology

3.1 Goals

The work described in this paper started from the need of implementing a garbage collector for a Java Optimised Processor (JOP) system. Our JOP version is a three stage pipeline, stack oriented architecture. The first fetches bytecodes from a method cache and translates them to addresses in the micro-program memory. The method cache is updated on invokes and returns from an external memory. The second stage fetches the right micro-instruction and executes micro-code branches. The third decodes micro-instructions, fetches operands from the internal stack, and executes operations. Due to its organisation, JOP can execute certain Java bytecodes as single micro-instructions, while more the complex ones as sequences of micro-instructions or Java methods.

The goal evolved from implementing a software solution towards designing a hardware garbage collection unit, that can operate concurrently with the application. In addition, we wanted to achieve a modular and scalable solution, independent of the main memory (heap) size. Overall, we wanted an architecture suitable for resource-limited embedded, and possibly real-time, systems.

3.2 Approach

Adding a hardware garbage collection unit into an existing Java system that had none at all initially, involves a number of changes and additions that have to fall into place. The class run-time images have to be augmented with GC information, which means modifying the application image generator. Certain bytecodes need to be modified to use the GC structures. The GC algorithm, the

hardware GC unit that implements it, and concurrency support have to be all correct. Errors would be hard to identify and debug in such a system. Therefore we adopted a step-by-step approach for developing the GC unit. We started by implementing a pure software mark-compact, stop-the-world GC algorithm (see Sect. 6.1). In the next step, we wrote the hardware GC unit, simulated, synthesised, and tested it in an artificial environment, were we could generate and control the memory contents at word level (see Sect. 6.2). In the final step, we optimised the GCU and customised the JOP core in order to integrate the hardware GC solution with the Java platform in an efficient way (see Sect. 6.3). The first phase was thus dedicated to check the correctness of the class level GC information. The second phase focused on designing, implementing and partially evaluating the GC unit. The final phase addressed the integration of the GC unit with the JOP-based system, involving changes both in the processor and GCU, as well as in their communication mechanisms.

4 The GC Algorithm

At the base of our implementation resides a typical mark-compact algorithm (see [11] for a brief description). Our software implementation uses a *stop-the-world* collector, which preempts the mutator when a *new* requests more memory than the available contiguous free space. For the hardware unit, we adopted an incremental version of the aforementioned algorithm (see [11] for details on incremental GC). Using the *tricolor marking* abstraction [7], in the marking phase, grey objects are maintained into a specific *GC stack*. Every write access to an object will cause the mutator to push its handle onto the GC stack, as it might have been altered and needs to be re-scanned. Concurrently, the collector extracts grey objects (handles) from the stack, marks them, scans them for references and pushes all the unmarked handles it encounters into the stack. The marking phase finishes when the stack becomes empty.

In the compacting phase, the heap is scanned with two pointers, *ScanPtr* is used to examine successively all the objects in the heap, while *CopyPtr*, trailing behind, identifies the end of already compacted objects. As soon as a marked object is found by the *ScanPtr*, it is copied at the *CopyPtr*, its handle updated, and both *ScanPtr* and *CopyPtr* are advanced. If non-marked objects are encountered, their handle is recycled and only *ScanPtr* is advanced.

In this phase, the interaction between the mutator and collector becomes a bit more complex. The situations that we want to avoid by proper synchronisation are those in which either a write access is made on a partially copied object or a read access is made on an object about to be overwritten. The first situation appears because the write access might modify an already copied word, that would need to be recopied to maintain coherency. For the second situation consider the following scenario. The mutator translates a handle to an instance pointer, meanwhile the collector moves the object and updates the instance pointer, and furthermore moves more objects until it overwrites the old copy of the first object. At this point the mutator holds a pointer to an invalid

location. To avoid such situations we introduced read/write barriers in the form of object-level locks, as follows.

Whenever the mutator performs a read access on a certain object, it must set a read lock on that object's address, and reset the lock (unlock) when the access is completed. If the collector is about to overwrite the read-locked address, it stalls until the lock is reset. The write lock mechanism is rather similar. Whenever the mutator intends to do a write on a certain object, it must set a write lock on that object's handle, and unlock it when the access is completed. If the collector is copying or about to copy the object in question, it stalls its execution until the lock is reset. At this point, however, all the progress is reset, and copying resumes from the beginning of that object. Note that in principle we could have chosen instead to make the mutator wait for the object to be completely copied, if the write access occurs as the object is being copied. However, in that case the interference with the mutator would have been too significant, and decided to rather have the collector do more work than having the mutator wait. The drawback of this solution is though the fact that GC progress is not always guaranteed, as detailed in Sect. 7.

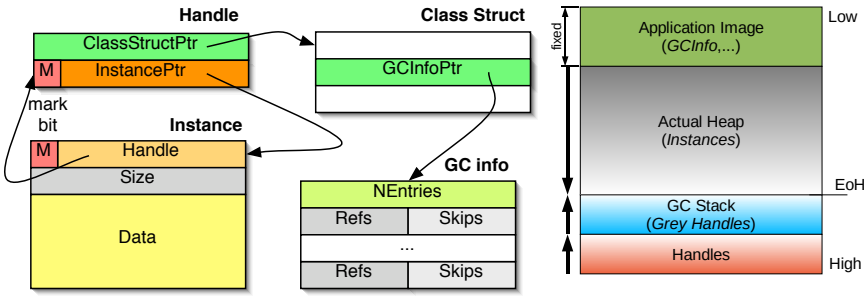
It is also important to notice that locking and unlocking are intended to occur at bytecode level, such that mutator threads cannot be preempted by other mutator threads while holding a lock. In fact this can be easily achieved in a JOP-based system, as it requires modifying the bytecodes for object accesses to also set and reset locks. This also means that only one lock can be set at any one time, observation that simplified the hardware architecture of the GCU.

5 Implementation Invariants

5.1 Data Structures

In a compacting GC algorithm, every time an object moves, all the pointers to that object would need to be updated. To overcome this we use *handles*, instance identifiers that are unique and constant throughout the object lifetime (see Fig. 1(a)). All accesses must first read the content of the object handle to find out the actual location of an instance.

Each instance header includes the size, the associated handle, and a mark bit. As instances are aligned to words, two extra bits are available in each pointer to an object – and we use one of them as a *marked* flag. Each handle is a pair of pointers, back to the instance and to the class structure, containing the necessary garbage collector information, *GCInfo*. The GCInfo structure is an array of pairs of half word values, the first containing the number of consecutive words that are references, and the second holding the number of consecutive words that are not references in the instance. A similar method of encoding information about the location of the references is used in [14]. The unused handles are maintained as a linked list, where the instance pointer is in fact the next free handle. Handles recycled through GC are added at the tail of the list, while handles for new instances are acquired from the head of the list.



(a) Object structure (b) Address space

Fig. 1. Implementation choices

5.2 Address Space Organisation

The memory in the GC-enabled system is organised basically in two parts, the application image and the heap (see Fig. 1(b)). The application image contains the class information, methods bytecode, static members, and constants, including *GCInfo* (see Fig. 1(a)). The heap is divided in three contiguous regions. The first contains the instances, and is in fact the actual heap. The second, situated at the end of the address space, holds the object handles. Whenever the application runs out of handles, meaning that the free handles list becomes empty, a new batch of handles is built toward lower addresses, shrinking the actual heap space. This space is never returned to the heap, but will be used as handles for the rest of the application life time. Finally, the third region, the *GCStack*, is only temporary. This extends between the handle space and grows downwards over the actual heap when the GC algorithm is in its marking phase. *GCStack* is in fact a handle stack used by the breadth-first traversal and marking of the live objects (see Sect. 4).

6 Design Iterations

Going from a system without any GC support to a concurrent hardware GC involves a significant number of additions and modifications. The GC information for each class has to be added in the application image, which means modifying the image generator. Locking mechanisms have to be added for the bytecodes accessing objects, which for JOP meant changes in the microcode. The GC unit had to be built and tested properly, if possible on a rather realistic setup, before adding it into the JOP-based system. Finally, the system needed to be optimised and adapted together with the GCU, which involves changes both in the GCU, the main processor (JOP), and communication structure. All these steps were gradually taken, in order to detect and tackle problems more efficiently.

6.1 Software, Stop-The-World on the Target Platform

At first, we implemented the GC in software on a system containing a custom version of the Java Optimised Processor (JOP, [5]). In this version, a GC cycle is performed in a stop-the-world manner, whenever a *new* cannot be performed because of the lack of free memory. Whenever this happens, the following steps are taken. The JOP stack is scanned for handles (root references), which are pushed into the GC stack. The MARK phase starts at this point. Live objects are traversed in a breadth-first manner, using the GC stack to store detected but not scanned handles. Once the GC stack is emptied, the COMPACT phase starts. Marked objects are compacted at the beginning of the heap and the marked flag cleared. The cycle terminates once all the heap objects are scanned. At this point *new* resumes its normal operation, by allocating the required heap space. Handles are also managed inside the *new* operation. Besides writing the actual GC code, a number of issues must be addressed.

Identifying Handles Initially, faithful to the JVM specification, JOP could not distinguish between handles and values on the stack. However, this is a must in order to register root references. The problem of identifying references is addressed for example in [15]. One solution is to use two separate stacks, one for references and one for values [14]. Another solution would be to tag each stack word with an extra bit, signifying reference. However, to avoid altering the JOP data path and the micro-instruction, we adopted the following, conservative method. As the handles are usually stored in a dedicated area, one can assume that any stack word is a reference if and only if points inside the handle area.

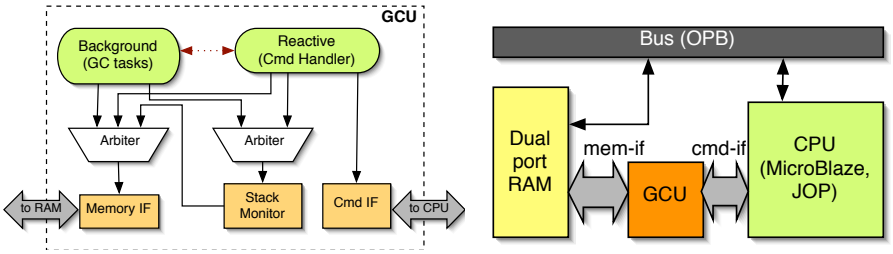
Bytecode Modifications Bytecodes in JOP are implemented either as micro-programs and/or as Java functions. To support our garbage collection solution, bytecodes accessing objects were modified to translate handles into references.

Application Image Impact Finally, the application image was extended with GC functionality (two classes) and augmented with the information necessary during the garbage collection cycle (GC info in Fig. 1(a)). The images increase in size by about 350 words (14%) including the GC required class structures and methods. This increase is marginally dependent on the number of objects, as each class is extended with 3-5 words of GC information. Certainly, the application image generator can be further improved to reduce these images even more.

Experimental Evaluation All of the systems used for evaluation in this paper were synthesised and run at 25MHz on a Xilinx Spartan2e (XC2S600e) FPGA. At this point we were mainly interested in the correctness of the GC implementation rather than in its performance. Nevertheless, measurements on similar benchmarks as the ones used later on in Sect. 6.2 revealed that a GC cycle takes in the range of tens of milliseconds, which is expected, considering the low CPU performance. This is yet another reason for exploring a hardware GC.

6.2 Hardware, Concurrent on a Test Platform

A concurrent collector should be able to both carry out the garbage collection and handle commands from the application at the same time. Let us name these tasks the *Background*, performing normal GC operations and the *Reactive*, handling commands. The *Reactive* would be required at points to access the GC stack, in order to store root pointers or grey references. The *Background* would also need to access the GC stack to push and pop handles as it marks live objects. Furthermore, stack operations should be atomic. It becomes apparent that a common resource, a *Stack Monitor*, used by both processes, is the solution. Additionally, the two processes and the stack monitor would in fact access the system memory. The architecture we decided to implement is depicted in Fig. 2(a). To make the design easier to port, we decided to use a single, common, memory interface (*Memory IF*) for all the accesses to the system memory. This can be easily rewritten to support all kinds of memory access protocols. Similarly, the interface through which GCU receives commands (*Cmd IF*) can be adapted to support various kinds of processor interfaces. All accesses to common resources are handled by arbiters using fixed priorities. The *Reactive* process has the highest priority, stack operations have medium priority, while the *Background* process has the lowest priority. In fact all modules from Fig. 2(a), except the arbiters, are synchronous (FSMs) exchanging values through hand-shaking. Most of the time the *Reactive* and *Background* processes synchronise and communicate indirectly through stack and memory accesses. However, for some commands (such as read/write lock, unlock, and new) the two processes communicate directly through hand-shaking, as the *Reactive* process must acknowledge the execution of these commands to the main processor. The modularity of the design, although very easily adaptable to various memory systems and processors, is also its downfall, as detailed in Sect. 6.3.



(a) Internal architecture (b) Placement in the system

Fig. 2. Hardware garbage collection unit

System Overview The placement of the GCU inside the complete system was chosen to minimise the interference with the CPU, taking advantage of the multi-port memory normally available on FPGAs. The generic solution is depicted in

Fig. 2(b). GCU is directly connected to the CPU (*cmd-if*), and to a second port of the memory architecture (*mem-if*). This way of integrating the GCU into the system will remain unchanged throughout our design process. Furthermore, most components around the GCU also remain fixed, as our CPU uses the On-chip Peripheral Bus (OPB) as system bus.

In order to evaluate our solution in a realistic but highly controllable environment, we connected the GCU into a MicroBlaze-based system. As command interface we used a dual Fast Simplex Link (duplex FSL), while the memory interface connects directly through a Block RAM port. Thus, we had a finer grain control over the memory structures used by the GC, as we used C written code to emulate, construct and verify object images. We could also use tested components and interfaces around the GCU, taking advantage of the better development tool support for MicroBlaze. Finally, we also wanted to achieve a rather processor independent design, portable to other systems.

Runtime Perspective The commands implemented initially by the GCU are gathered in Table 1. From the programmer's point of view, the concurrency-related commands are *rdlock*, *wrlock*, *unlock*, and *waitidle*. The first three are used to ensure the correct access to objects, while the last is used to join the GC process with the main application. Some of these commands are synchronous, causing the GCU to send an acknowledgement (*GCU Ack* column), while others can be just issued without expecting a reply from the GCU.

Table 1. Initial GCU Commands

Command	Words	GCU Action	GCU Ack
[rd,wr]lock h	1	Locks object with handle *h*.	obj address
unlock	1	Unlocks a previously locked object.	yes, any
waitidle	1	Waits until GCU is IDLE.	end of heap
stackinit a	2	Sets GC stack base at *a*. Starts MARK phase.	none
rootref h	2	Registers *h* as root reference.	none
docmpct h	2	Starts COMPACT. Freed handles appended to *h*.	none
new h, s	2	Creates an object of size *s* and handle *h*.	end of heap

A GC cycle is triggered by the CPU, by issuing a *stackinit* command to the GCU (see Fig. 3.a), which goes from IDLE to MARK phases. The CPU registers root pointers by pushing them in the GC stack via *rootref* commands. Concurrently, as soon as the GC stack contains handles, the GCU starts marking objects. Once all the root references have been pushed into the GC stack, the CPU issues a *docmpct* command, allowing the GCU to start the COMPACT phase. Next, the CPU can start executing application code, using *rdlock*, *wrlock*, *unlock*, and *new* as in Fig. 3.b. As soon as all the live objects have been marked (the GC stack becomes empty), the GCU begins the COMPACT phase. The CPU can wait for the GCU to finish by issuing a *waitidle*, update the end of heap received from the GCU, and maybe start a new GC cycle.

Inside the application code, every time a new object is created, the CPU must issue a *new* to the GCU and wait for an acknowledgement (see Fig. 3.b). Note that acquiring a new handle and determining the object size is the responsibility of the software *new object* function or micro-code. Accesses to objects data (GET/PUTFIELD, *ALOAD, *ASTORE, ...) must be preceded by *rd/wrlock* on the object handle. The CPU must wait for an acknowledgement of the lock from the GCU before translating the handle into a real address and accessing the object. The CPU must call *unlock* as soon as the access is completed.

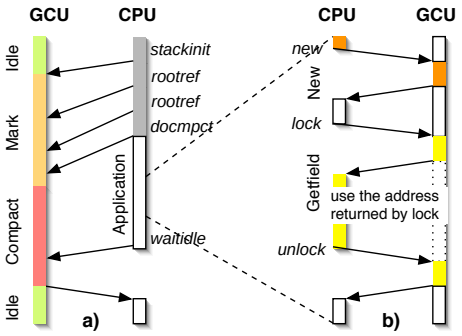


Fig. 3. GCU-CPU synchronisation: a) A GC cycle, b) New and the lock mechanism.

Experimental Evaluation We initially compared the performance our hardware approach to a software version of the same algorithm, both used in a stop-the-world manner. The software version was coded in C and compiled with *gcc* using the highest optimisation level. The mutator was a simple application that creates alternatively elements from two linked lists, and keeping both, one or neither of the lists alive followed by a GC cycle. The three situations reflect three different memory configurations, when no objects are moved, half of the objects are moved and finally when all objects are discarded. The GC cycles were timed using an OPB_TIMER core. The results for lists of ten, thirty, and fifty elements are depicted in Fig. 4. Note that the GCU performs consistently around four times faster than the software version. Some of the speed-up is due to using a memory port directly instead of the OPB, as in the software version. However, the GCU used in this experiment is not optimised as a stop-the-world version, but instead implements all the features required for concurrent GC.

Looking at the overhead introduced by synchronisation into the object access latency, one lock/unlock pair takes about 19 clock cycles altogether. This is a much larger overhead than we initially intended. The reason behind this large overhead lay in the architecture of the GCU which was constrained by the capabilities of the MicroBlaze core. In particular, for the lock-related commands, they need to propagate through the FSL, are decoded in the *Reactive* process,

and finally reach the *Background* process. Furthermore, an acknowledgement follows the same lengthy path back to the processor. Using a dedicated channel only for the locked address or handle, directly from the processor to the GCU, the latency would reduce considerably. This would imply modifying the processor core, which is impossible for proprietary IP such as MicroBlaze. However, for the GCU version intended for JOP (an open core) this is not a problem.

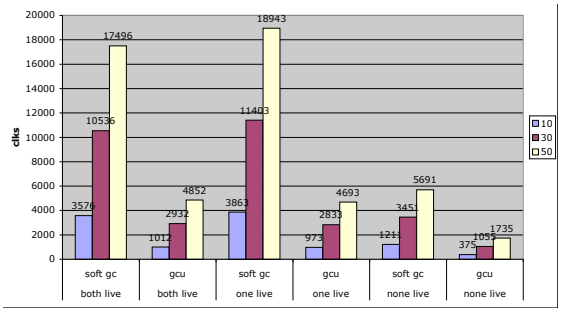


Fig. 4. Stop-the-world GC latency of the GCU vs. a software, C version (soft-gc) for an application using two linked lists with 10, 30, and 50 elements alternatively created.

6.3 Improved Hardware, Target Platform Specific

Integrating the GCU with the target platform, centred around JOP, imposed and at the same time allowed for a number of changes in the CGU command and memory interface. In this iteration the system used a standard Local Memory Bus (LMB) for memory interface, and a custom command interface (called Fast Command Access – FCA), while the rest of the architecture remained basically the same (see Fig. 2(b)). Micro-architectural changes in JOP and also GCU had to be implemented in order to employ the specialised FCA. Compared to the pure software approach, there has been migration of the software functionality into hardware, which affected both the application image and the device utilisation.

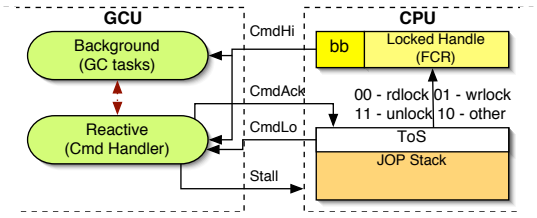


Fig. 5. Fast Command Access interface: a deeper integration of the GCU and CPU.

Fast Command Access Interface. The FCA was designed as an alternative to the FSL, in order to reduce the lock and unlock overhead. The JOP core was extended as follows. First, a *fast command register* (FCR) was introduced inside the processor, register directly seen by the GCU (*CmdHi* in Fig. 5). This register holds the currently locked handle, value that needs to be persistent while the lock is in effect. To allow extended commands (double word) to reach the GCU, the top of the stack from JOP is also made visible to the GCU (*CmdLo* in Fig. 5). The GCU can also send back to JOP words through a *CmdAck* signal. Furthermore, the GCU can stall JOP, if needed, through a *Stall* line. The JOP micro-instruction set was extended with two new ones: **stfc** pops the stack and stores the value into the FRC, and **ldfc** pushes *CmdAck* into the stack.

The GCU uses the FCA interface as follows. Read, write locks, and unlocks require one word (*CmdHi*) for identification. The *Background* process can readily use handles stored in *CmdHi* to determine lock situations. At the same time, the *Reactive* process decodes both *Cmd* lines and drives the *CmdAck* when necessary. In particular, read/write lock latency is reduced to one memory access, needed by the *Reactive* process to translate the object handle into an address.

Application Image Impact. The software GC functionality is now obsolete, being implemented by the GCU. The application image has been reduced by approximately 100 words compared to the pure software implementation.

Device Utilisation Impact. Adding the GCU doubles the area used for the JOP system (see Table 2), which includes along with JOP itself an OPB bus connecting a Block RAM, a UART, a timer, and some general purpose I/O cores. The synthesis tool reports the GCU clock frequency at 78MHz on the XC2S600e.

Table 2. GCU, JOP, and systems device utilisation (on a Xilinx Spartan2e, XC2S600e)

Unit resources	GCU only	JOP only	**Full system resources**	JOP, RAM, IPs	JOP, GCU RAM, IPs
Slice FF	900	400	Slices	1543 (22%)	3053 (44%)
4LUT	2966	1783	BRAMs	71 (98%)	71 (98%)

Experimental Evaluation. Using a JOP-based system, we evaluated the performance of the GCU as a stop-the-world garbage collector, with the same application from Sect. 6.2. As expected, the figures were similar to the ones reported for the MicroBlaze test system (see Fig. 4, GCU), as only the interface to the GCU changed. Compared to the pure Java GC, the speed-up achieved by using the GCU is impressive, a GC cycle being almost a hundred times shorter.

Next, using ModelSim, we examined the overhead introduced in JOP by the synchronisation required for some of the GCU commands, namely *(rd/wr)lock*,

unlock, and *new*. The locking mechanism is employed to make sure that an object does not move during an access, while *new* is needed for allocating new objects. The overhead varies between 2 and 7 clock cycles for lock related operations and 15 clock cycles for *new*. Note that these figures reveal a lower overhead for locking and unlocking objects compared to the MicroBlaze-tested solution (Section 6.2), due to its improved GCU-CPU communication. However, it makes more sense to look at this overhead in the context of bytecodes, as all those using references need to employ the locking mechanism. In particular, the micro-program associated with bytecodes reading(writing) object contents needs to be extended with read (write) locks on the object handle before the access and conclude with an unlock. As each bytecode is implemented as a sequence of micro-instructions, the actual overhead of the locking mechanism is even smaller at this level. For bytecodes implemented as Java methods by JOP (NEW, NEWARRAY, ANEWARRAY), the overhead becomes negligible (see Table 3). The most strongly affected is PUTFIELD, that increases its execution time by 50%. Nevertheless, the impact these have on the applications overall, depends highly on the specific application.

Table 3. Maximal synchronisation overhead in clock cycles, per bytecode class.

read access bytecodes				**write access bytecodes**			
class	latency (clock cycles)			class	latency (clock cycles)		
	before	after	change		before	after	change
GEFIELD	28	31	11%	PUTFIELD	30	45	50%
*ALOAD	41	44	7%	*ASTORE	45	60	33%
ARRAYLENGTH	15	18	20%	NEW,	Java		
INVOKE*	> 100	+3	$< 3\%$	*NEWARRAY	methods		$< 1\%$

7 Discussion

Ensuring GC Progress One of the goals we set for our GC solution is very low interference with the application, which translates into low latency for accessing objects. In other words, *lock/unlock* and *new* must be as fast as possible. Furthermore, deterministic times for these operations are also desired for real-time applications. Although these goals have been achieved, there is a price to pay on the GC side. In particular, write locks/unlocks in the middle of an object move force a rollback of the progress to the beginning of that object. For large objects, frequently written, it could happen that the GC will never be able to finish moving the object in question. This can happen if the time between two write accesses of an object is shorter than the time needed to move the whole object. Nevertheless, there are several ways of ensuring progress in such cases. Offline analysis can reveal the maximum size of an object for which the GC still makes progress. One can then either rewrite the application code or split

the objects into smaller objects. Another possibility would be a time-out behaviour, delaying the application when the GC makes no progress for a certain time interval.

Another Processor as GCU For more flexibility, one can imagine using a second processor instead of a hardware GCU. The initial tendency is to implement the *Reactive* process (see Sect. 6.2) as an interrupt handler, leaving the *Background* process as the normal operation mode. However, the interrupt handling latency for a general purpose processor is at least in the range of tens of clock cycles, making the lock/unlock latency too large to be useful. Nevertheless, with the advent of reactive processors, such as ReMIC [16], the possibility of a processor-based GCU appears feasible and exciting.

Real-Time Considerations Standard garbage collectors are allocation triggered, meaning that whenever the free memory decreases under a certain limit, a GC cycle is started. The GCU we presented in this paper may very well be employed in that manner. However, our GCU is more suited for a *time-triggered garbage collection* (TTGC) approach, offering better real-time performance [17]. As GC in our approach is truly concurrent, it makes more sense to view a GC cycle as a task, rather than as small increments packed into allocation steps. This task is then treated no differently than the rest of the tasks in a real-time system, as long as it can be performed often enough to provide the necessary free memory. Theorem 1 in [17] provides an upper bound for the GC cycle time that guarantees that there will always be enough memory for allocation. In our case, this is the period our GCU must be initialised and allowed to run a full GC cycle. However, having the CPU and GCU synchronise now and then via locks introduces additional delays.

Another way to employ the GCU would be to run it constantly, staring a new cycle as soon as the current one finishes. However, depending on the application, this may lead to a large amount of wasted work and energy.

8 Conclusion

The current paper presented a hardware garbage collection unit, designed to work concurrently with the CPU in Java-based embedded system. Given the complexity of adding a concurrent GC into a system without any GC support at all, a gradual design approach was taken, to identify and fix problems easier.

To satisfy the requirements of minimal interference GC, our solution involves not only an efficient GC unit, but also specialised support in the processor, necessary for fast CPU-GCU interaction. The dedicated hardware GCU itself consists of two processes, one dedicated for handling commands and synchronising with the CPU, and the other implementing a mark-compact GC algorithm.

As a stop-the-world garbage collector, our GCU is four times faster than a highly optimised C solution, and orders of magnitude faster than a Java solution.

As a concurrent solution, the locking mechanism introduced to keep memory consistency introduces a small overhead in the system, bringing the benefit of running in parallel with the application. Finally, our solution seems to be suitable for real-time applications, when time-triggered garbage collection is employed.

References

1. Sun: PicoJava-II microarchitecture guide. Technical Report 960-1160-11, Sun Microsystems (1999)
2. Hardin, D.S.: aJile systems: Low-power direct-execution Java microprocessors for realtime and networked embedded applications. (aJile Systems Inc.)
3. : Moon2- 32 bit native Java technology-based processor. (Vulcan Machines Ltd.)
4. : Lightfoot 32-bit Java processor core. (Digital Communication Technologies)
5. Schoeberl, M.: JOP: A java optimized processor. In: Workshop on Java Technologies for Real-Time and Embedded Systems. (2003)
6. Steele, G.L.: Multiprocessing compactifying garbage collection. Communications of the ACM **18** (1975) 495–508
7. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM **21** (1978) 966–975
8. Baker, H.G.: List processing in real-time on a serial computer. Communications of the ACM **21** (1978) 280–294
9. Boehm, H.J., Demers, A.J., Shenker, S.: Mostly parallel garbage collection. In: Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation. (1991) 157–164
10. Printezis, T., Detlefs, D.: A generational mostly-concurrent garbage collector. In: Proceedings of the ACM SIGPLAN International Symposium on Memory Management. (2000) 143–154
11. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proc. Int. Workshop on Memory Management, Springer-Verlag (1992)
12. Schmidt, W.J., Nilsen, K.D.: Performance of a hardware-assisted real-time garbage collector. In: *Proceedings of the Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. (1994) 76–85
13. Srisa-an, W., Lo, C.T.D., Chang, J.M.: Active memory processor: A hardware garbage collector for real-time java embedded devices. IEEE Transactions on Mobile Computing **2** (2003) 89–101
14. Ive, A.: Towards an embedded real-time Java virtual machine. Lic.Thesis 20, Dept. of Computer Science, Lund University (2003)
15. Agesen, O., Detlefs, D.: Finding references in java stacks. In: OOPSLA'97 Workshop on Garbage Collection and Memory Management. (1997)
16. Salcic, Z., Hui, D., Roop, P., Biglari-Abhari, M.: ReMic - design of a reactive embedded microprocessor core. In: Proceedings of Asia-South Pacific Design Automation Conference. (2005)
17. Gestegard-Robertz, S., Henriksson, R.: Time-triggered garbage collection. In: Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems. (2003)