FPGA-Aware Garbage Collection in Java

Philippe Faes, Mark Christiaens, Dries Buytaert, Dirk Stroobandt
Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
email: {pfaes,mchristi,dbuytaer,dstr}@elis.ugent.be

Abstract— **During codesign of a system, one still runs into the impedance mismatch between the software and hardware worlds. This paper identifies the different levels of abstraction of hardware and software as a major culprit of this mismatch. For example, when programming in high-level object-oriented languages like Java, one has disposal of objects, methods, memory management, ... that facilitates development but these have to be largely abandoned when moving the same functionality into hardware.**

As a solution, this paper presents a virtual machine, based on the Jikes Research Virtual Machine, that is able to bridge the gap by providing the same capabilities to hardware components as to software components. This seamless integration is achieved by introducing an architecture and protocol that allow reconfigurable hardware and software to communicate with each other in a transparent manner i.e. no component of the design needs to be aware whether other components are implemented in hardware or in software.

Further, in this paper we present a novel technique that allows reconfigurable hardware to manage dynamically allocated memory. This is achieved by allowing the hardware to hold references to objects and by modifying the garbage collector of the virtual machine to be aware of these references in hardware. We present benchmark results that show, for four different, well-known garbage collectors and for a wide range of applications, that a hardware-aware garbage collector results in a marginal overhead and is therefore a worthwhile addition to the developer's toolbox.

I. Introduction

Hardware/Software codesign has become an indispensable tool in the designer's toolbox for building advanced embedded systems. It provides the designer with the best of both worlds: software (SW) development lends itself very well to building components that have a complex control-flow while hardware (HW) development excels at exploiting the maximum amount of parallelism that is present in the application and therefore achieves great speed-ups.

There is a catch: the SW and HW worlds speak totally different languages. A SW developer thinks in terms of data structures (preferably even object-oriented) and methods that operate on them. These methods are constructed using control-flow constructs such as loops, tests, method-calls, ... The SW developer's temporal view of the execution is often that of a purely sequential execution or maybe an execution with a mild amount of parallelism (using threads or message passing systems).

A HW developer on the other hand works with a model of the application that is much more physically oriented. Data resides in memories and registers and is operated upon by combinational logic built up from logic gates. Execution is often extremely parallel and is modelled by hundreds of parallel processes that function simultaneously.

All this results in a serious impedance mismatch when trying to unite the two worlds through codesign. A number of attempts to address these issues have already been made [1], [2], [3]. We focus on the interface between SW and *reconfigurable* HW because of reduced cost and ease of prototyping and because reconfigurable HW allows for time-multiplexing [4].

One cannot fail to notice that existing codesign support is still in its infancy so in [5] we proposed a codesign methodology with full support for object-oriented design and which bridges the gap between HW and SW in a much more elegant way. We presented a Java [6] runtime environment where complete transparency between a HW or SW implementation of tasks is aimed for i.e. where HW tasks have essentially the same capability to call methods, create objects, ... as their SW counterparts and can therefore readily be interchanged.[1] In addition it provides a shared-memory model (as opposed to message passing [7], [8] or remote procedure calls with marshalling [9] used in existing methodologies) where parallel executing tasks are modelled as threads communicating through read, write and synchronisation instructions on a shared memory.

Until recently our methodology only supported HW components which do not hold references to objects i.e. which do not have any impact on dynamic memory allocation. All dynamic memory management was performed by the SW acting as a proxy on behalf of the HW.

In this paper we present novel work enabling a HW component to actually create *and hold references to* dynamically allocated objects. Since HW components can now hold objects, the Java garbage collector needs to be made aware of the existence of these object references in order to find all objects that are still in use. In addition, most Java environments will perform a compaction phase after the garbage collection phase in order to create a contiguous area of free memory. To support compaction we enable the garbage collector to actually update references to objects residing in HW. Our techniques have been validated for four different garbage collectors and have been tested on benchmarks from the DaCapo Benchmark Suite [10].

The rest of this paper is organized as follows. First we explain the rationale of passing object references to HW. In Section 3 we describe the architecture and the protocol we

[1]The decision between HW and SW is made by the runtime environment. E.g. when the HW is executing one invocation of a method, another invocation can be executed in software.

used for passing these references. Section 4 explains in depth a use-case of a method invocation as an example of how the protocol works. Section 5 discusses how it is possible to make garbage-collector-safe HW, and how we adapted the garbage collector for HW. The validation and some measurements of our implementation is presented in Section 6. Finally we conclude and discuss our planned future work.

II. Rationale of Passing Object References to HW

In [5] we showed how HW components could be fit transparently into the execution of a Java program on a Java virtual machine (JVM). A shortcoming of this approach is that HW components could not receive or hold actual references to objects. All operations related to objects are performed through the intermediation of the SW. Nevertheless, it is very useful to grant the HW the same capabilities as the SW with regard to references to objects. We discern three advantages.

First, the number of HW registers used for parameter passing during a method call to HW can be limited. Instead of passing a large amount of parameters to the HW we can simply provide a reference to an object that contains (or can reach) the required data.

Second, neither the SW nor the HW needs to know in what type of memory the object actually resides. A reconfigurable computing device can contain different memories, some of which are tightly coupled with the CPU, while others are more tightly coupled with the HW. The JVM runtime can implement heuristics that place data closer to the computing unit that is likely to use the data in the near future. These heuristics can be based on profilation data or programmer hints.

Finally, calling a method, whether it is implemented in SW or in HW is completely transparent. Both a SW or HW method will receive the same parameters. This allows a HW method to perform the same operations as a SW method: it can call methods of any of its arguments or pass its arguments on to other methods, independent of whether or not these methods are implemented in HW or SW.

We will now introduce the protocol we use for calling HW methods from SW and SW methods from HW with full support for reference handling.

III. Architecture and Protocol

We model a reconfigurable computing device as a shared memory machine. The address space of the HW is mapped into the address space of the JVM, therefore the JVM can access memory and control registers of the HW. The HW can also access the JVM's heap, e.g. through Direct Memory Access (DMA).[2]

Fig. 1 shows the memory map of a HW block, as seen by the JVM. The four boxes on the left represent no physical HW, but rather address ranges on the bus. Addresses that can be read by the JVM are indicated with an R, writable addresses with a W. These address ranges are mapped to either internal RAM or internal registers of the HW. The *garbage collection region* (range 0xc00 to 0xffc) has a one-to-one mapping to the internal RAM for object references. *Method parameters* (0x400 to 0xbfc) and MethodReturnValue locations (0x00c, 0x010, 0x020, 0x024) are mapped arbitrarily onto addresses in the internal RAM for primitive datatypes or the internal RAM for object references.

The remaining addresses in the *control registers* region are not mapped onto RAM, but are either ignored or directly connected to registers in the HW. Data with a 32-bit size can be passed to the HW by writing into a register. These registers are marked word. Simple signals (indicated with bit-W) can be sent by writing to the corresponding address. When this happens, a single bit signal will be set to high during one clock cycle on the HW side. Addresses indicated with bit-R represent single bit output pins of the HW. If the HW drives a low signal, the JVM will read 0 at this address, otherwise -1 is read.

IV. Use-Case: Passing References Between HW and SW

In Sections II and III we have explained why it is important to pass objects by reference and which protocol we will use to pass these references. To illustrate this protocol, we will give an example of a method invocation in HW.

Consider the method invocation of method1 in Fig. 2 (line 1) with parameters s, o and l. The Java run-time environment intercepts the method and decides it should be executed in HW. First the arguments are written to the HW; the address of object p is written to address 0x400, the value of s to 0x404 and the address of o to 0x408. Since l is of type long and requires 64 bits, the lower (least significant) 32 bits are written to 0x40c and the higher 32 bits are written to 0x410. Now the SW starts the HW by writing any value to 0x004. The Java thread now waits until it notices a request from HW. This request can be seen through polling or through an interrupt mechanism.

The HW has its address decoder configured as indicated in Fig. 1, so that its arguments p, s and o are written to locations REFS0, PRIM0, REFS1, respectively. The 32 least significant bits of l are mapped on PRIM1 and the 32 most significant bits and PRIM2 respectively. The computation starts and at some point it wants to call method2 (line 3). The value of i was calculated by the HW (line 2) and stored in PRIM3. Since method2 is a static method, there is no this[3] object, so the value of this is set to null. The HW reconfigures the address decoder so that address 0x800 is mapped to a memory location that contains the value 0x0, indicating a null pointer. The parameters i and o are set by mapping addresses 0x804 and 0x808 to PRIM3 and REFS1. The value to be read at 0x018 is set to an integer that identifies method2. Now the SW is signalled by setting the SWMethodRequest pin to 1; from now on the SW will read 0xffffffff at address 0x014.

The SW notices that the HW wants to call a method. It reads the value at 0x018 and looks up which method is indicated by that value. Knowing which method will be invoked, the runtime environment knows that only two word-sized arguments need to be read from HW. Both arguments are read at 0x804 and 0x808, respectively. The method2 is now invoked. Since the

[2]This requires that the caches are flushed before invoking HW and invalidated when HW returns control to SW.

[3]The Java keyword this indicates on which object a method is called. Static methods are called without a corresponding object.

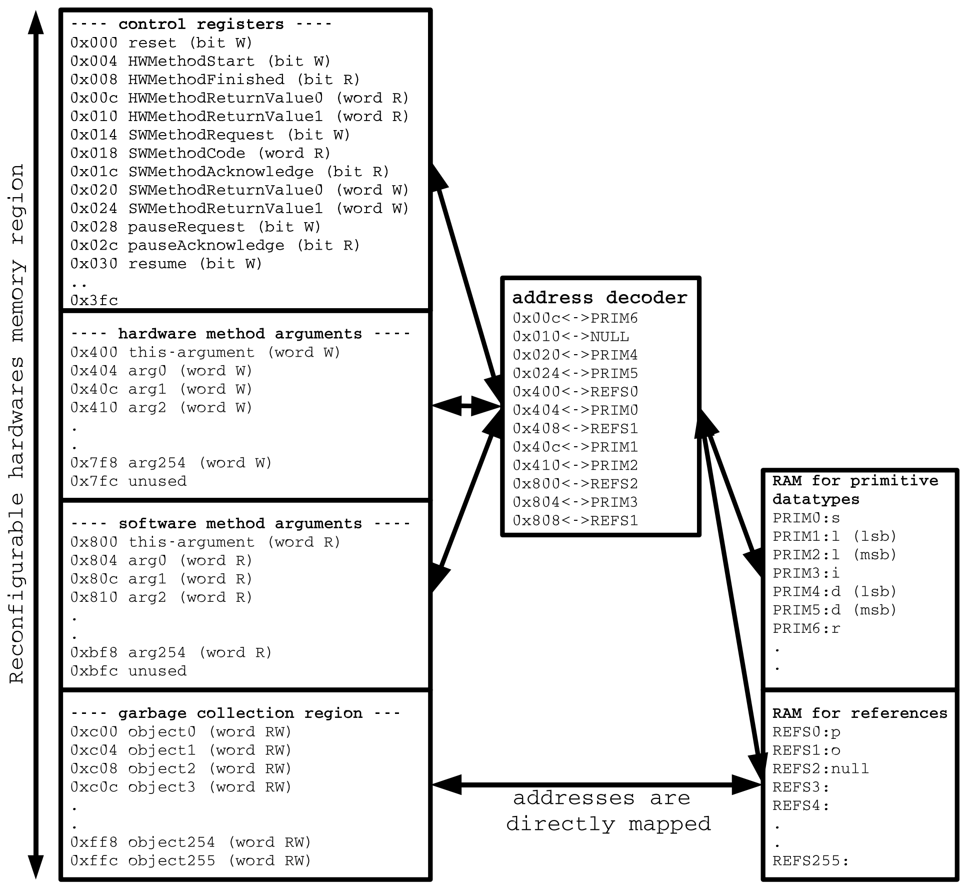
Reconfigurable hardwares memory region
---- control registers ----
0x000 reset (bit W)
0x004 HWMethodStart (bit W)
0x008 HWMethodFinished (bit R)
0x00c HWMethodReturnValue0 (word R)
0x010 HWMethodReturnValue1 (word R)
0x014 SWMethodRequest (bit W)
0x018 SWMethodCode (word R)
0x01c SWMethodAcknowledge (bit R)
0x020 SWMethodReturnValue0 (word W)
0x024 SWMethodReturnValue1 (word W)
0x028 pauseRequest (bit W)
0x02c pauseAcknowledge (bit R)
0x030 resume (bit W)
..
0x3fc
---- hardware method arguments ----
0x400 this-argument (word W)
0x404 arg0 (word W)
0x40c arg1 (word W)
0x410 arg2 (word W)
.
.
0x7f8 arg254 (word W)
0x7fc unused
---- software method arguments ----
0x800 this-argument (word R)
0x804 arg0 (word R)
0x80c arg1 (word R)
0x810 arg2 (word R)
.
.
0xbf8 arg254 (word R)
0xbfc unused
---- garbage collection region ---
0xc00 object0 (word RW)
0xc04 object1 (word RW)
0xc08 object2 (word RW)
0xc0c object3 (word RW)
.
.
0xff8 object254 (word RW)
0xffc object255 (word RW)
address decoder
0x00c<->PRIM6
0x010<->NULL
0x020<->PRIM4
0x024<->PRIM5
0x400<->REFS0
0x404<->PRIM0
0x408<->REFS1
0x40c<->PRIM1
0x410<->PRIM2
0x800<->REFS2
0x804<->PRIM3
0x808<->REFS1
RAM for primitive datatypes
PRIM0:s
PRIM1:l (lsb)
PRIM2:l (msb)
PRIM3:i
PRIM4:d (lsb)
PRIM5:d (msb)
PRIM6:r
.
.
RAM for references
REFS0:p
REFS1:o
REFS2:null
REFS3:
REFS4:
.
.
REFS255:
addresses are directly mapped

Fig. 1. Mapping for Garbage Collector

```
int method1(short s, Object o, long l){ // executed in HW
  int i = . . . ; // calculate i
  double d = method2(i, o);
  int r = . . . ; // calculate r;
  return r;
}
static double method2(int j, Object p){ // executed in SW
  . . .
}
```

Fig. 2. Example method call

return value requires 64 bits, it is split up in two parts that are written at 0x020 and 0x024. The SW acknowledges that the method call is completed and waits for another signal from HW. This is done by writing a value to 0x01c and waiting until the value at either 0x014 or 0x008 is 0xffffffff.

The HW can continue its calculations. Once all calculations are finished, it writes the final result r in PRIM6. Address 0x00c is then mapped to PRIM6. The HW now sets its HWMethodFinished pin, so that the SW can read 0xffffffff at address 0x008.

Because HW does not know when SW will read the return value, it keeps the data at 0x00c valid until SW starts a new HW method invocation or until the HW is reset by writing to address 0x000.

V. A HW-Reference Aware Garbage Collector

A. The Java Garbage Collector

Java has a system of automatic dynamic memory management. This means the programmer can allocate memory in an address space called the *heap*. This is done simply by creating new objects. The programmer, however, cannot free any allocated memory. Instead, this is done by a part of the run-time system called the *garbage collector*. Whenever the heap is full, the garbage collector eliminates objects in the heap that can no longer be reached by the Java program. These objects are called *dead* objects; objects that can still be reached by the Java program are *live* objects. Determining which objects are dead or alive is a problem of graph connectivity in the *object-reference graph*. This is a directed graph where all nodes represent objects and edges represent references from one object to the other.

In addition to removing unused objects, the run-time system can optionally move objects to one contiguous memory region of live objects, thus creating a contiguous region of unused memory. This is called *compaction* and is done in order to prevent memory fragmentation.

We will concentrate only on garbage collectors that stall the entire Java application while collecting garbage. This type of garbage collection is called *stop-the-world* garbage collection. Stop-the-world collectors assume they have full control over all object references, and that no object references are hidden from them. In other words, the object-reference graph is not altered by another process during garbage collection and the *entire* object-reference graph is visible to the garbage collector.

There are four important sub-goals in order to assure the requirements are met.

1) We need to pause the entire application so that the object-reference graph cannot be altered during garbage collection.
2) All object references present in the HW (e.g. passed to HW as method parameter or method return value) have to be exposed to the garbage collector.
3) All object references in Java objects (e.g. in containers or arrays, fields of objects) on the heap have to be exposed to the garbage collector.
4) When passing references between Java and HW, no object references may be hidden from the garbage collector in any communication channel, such as the system bus, or low-level SW routines.

Note that the first sub-goal ensures that the object-reference graph does not change during garbage collection, and sub-goals 2–4 ensure that the entire object-reference graph is visible to the garbage collector.

B. Pause / Resume

We have adapted the Jikes Research Virtual Machine (Jikes RVM, a virtual machine formerly known as Jalapeño [11] and almost completely programmed in Java) to achieve the sub-goals spelled out in the previous subsection. The first modification was to provide a list of all available HW entities to the garbage collector. Whenever the garbage collector is invoked, it will send a *pause* signal to each HW entity in the list. This is done by simply writing to the pauseRequest address (cf. Fig. 1) of each HW entity. When HW receives a pause request, it stops working and acknowledges the request through a memory mapped register. Once the garbage collector has checked that all HW entities are paused, it can proceed. After the garbage collector finishes, a *resume* signal is sent to HW.

Note that pausing does not mean that HW has to stop all computations. We only demand that no references are changed, so computations that only involve primitive data can continue.

C. References in HW

Whenever HW is paused, all of its object references have to be visible to the garbage collector. We provide a dedicated garbage collection region in the memory mapped address space of the HW. The garbage collector can read and update all of the HW's object references through this region.

The first implementation we propose will *always* expose all references to the garbage collector, whether the HW is paused or not. In this implementation all object references are held in an internal RAM for object references (Fig. 1). This RAM is directly mapped onto the garbage collection region. If the HW wants to receive an object reference as method parameters, it can map the corresponding parameter address to a location in the internal reference memory. A primitive parameter should never be mapped to the RAM for references, since the garbage collector would mistake it for an object reference. Instead it can be mapped to a separate RAM for primitive datatypes. The return values of methods are mapped onto internal RAM in the same manner as parameters.

As an alternative implementation, we fully exploit the pausing mechanism introduced in the previous subsection. In this implementation object references are not stored in a separate RAM that is constantly exposed to the garbage collector. Instead, object references can reside hidden inside the HW, e.g. in many different small RAMs or in registers. Whenever a pause is requested by the garbage collector, the HW needs to expose all of its object references through the garbage collection region. If needed the HW can take several clock cycles to copy all references from its internal registers and RAMs into the garbage collection region. When this operation is finished, the HW acknowledges the pause request and garbage collection can start.

After the garbage collector has finished and the HW has received a resume signal, it can copy the updated object references back into its internal registers and RAMs.

D. References in SW

Since the garbage collector is designed to trace object references in SW, there are no further steps required to accomplish this sub-goal.

E. References in Communication Channels

While passing an object reference between HW and SW, there is a short moment when the reference is just a sequence of bits, without the explicit semantics of an object reference. If SW passes a reference to HW, it first looks up the address of the object. This address is then written to HW through memory mapped I/O. We require all I/O to be completely ordered in time, in other words, every started I/O operation finishes completely before another I/O operation can be started.

In order to pass an object reference from SW to HW we need to look up the address of the object, and write that address to HW using memory mapped I/O.[4] This means that there exists a moment in time when we work on *raw addresses* instead of object references. The difference is that raw addresses are not automatically updated by the garbage collector, and consequently they are not safe.

To pass object references in a garbage-collector-safe fashion, the address lookup and the address write need to be performed *atomically*. This can be accomplished by (i) making sure both operations occur in the same *method* and (ii) making sure this method will not be interrupted by a thread switch or by the garbage collector. This is done using a compiler pragma[5].

[4]Both operations (looking up addresses and performing memory mapped I/O) are impossible in ordinary Java code, but Jikes RVM provides *magic* methods that do allow such operations.

[5]Pragmas are, again, not supported in regular Java, but they are in the Jikes RVM

VI. Performance measurements

We evaluate the techniques presented in this paper using four garbage collectors, respectively the MarkSweep, SemiSpace, GenCopy and GenMS collectors provided by MMTk [12]. The machine used for benchmarking is a Symmetric Multi Processor (SMP) machine that is equipped with two AMD Athlon MP 2 GHz processors with 512 KiB cache each and 1 GiB main memory. It runs a Linux 2.6.8 kernel with SMP support. The FPGA is an Altera Stratix 1S25 hosted on an Altera PCI development board. This board plugs into a 66 MHz PCI slot, and uses the PCI bus as 32-bit bus.

The virtual machine we used is an adapted version of Jikes RVM 2.3.2. We used the FastAdaptive just-in-time compiler and specified no further command-line arguments. This results in an initial heap size of 20 MiB and a maximum heap size of 100 MiB.

A. Micro Benchmark

In order to compare the cost of garbage collection with HW and garbage collection in a pure SW environment, we devised the following micro benchmark. First, the garbage collector is informed of the existence of n instances of HW. In normal operation these HW instances would be used to accelerate some Java methods, but we only measure the garbage collection cost without any actual HW acceleration. The HW does not do any computation, it only holds references and accepts pause and resume signals. Each instance can hold 256 object references to Java objects. We create n$\times$256 objects, and pass their object references to the HW instances. We then measure the duration of a garbage collection by forcing a garbage collector invocation in a loop and measuring the execution time of the loop. We also ran the same benchmark without passing the object references to HW. Instead the object references were held in an array of objects.

The execution times are given in Table I(a). The cost for collecting objects in HW raises linearly with the number of objects in HW. In full-heap collectors (SemiSpace and MarkSweep) the overhead for HW collection is negligible, except when a very high number of objects is referenced in HW (n=100 ; 25600 objects in HW). We expect that HW will be used only for speeding up a very limited number of methods, and will only hold a very limited number of objects, e.g. n=10; 2560 objects or less. In generational collectors (GenMS, GenCopy), the execution overhead also rises linearly, but since a first-generation garbage collection execution takes very little time, the HW-aware garbage collection cost is large compared to the original garbage collection cost.

B. DaCapo Benchmarks

The DaCapo Benchmark Suite [13] is a suite of programs for testing memory management systems. We used six benchmarks: antlr, bloat, fop, hsqldb, jython, and ps from version *beta050224*. Each benchmark was run without HW overhead, and with a HW overhead of 256, 2560 and 25600 object references in HW. These references refer to objects created before the benchmark actually started, introducing an *extra* load

TABLE I

Top table(a): Micro benchmark execution times (ms). Bottom table(b): Relative delay for 25600 objects in HW, compared to benchmarks without HW.

(a)	n	Semi-Space	Mark-Sweep	Gen-Copy	Gen-MS
extra	0	196.1	204.1	1.269	1.255
objects	1	196.1	204.8	2.193	2.074
in	10	200.7	209.6	5.805	5.712
HW	100	241.0	249.6	46.216	45.834
extra	0	195.9	205.0	1.293	1.267
objects	1	196.3	204.7	1.294	1.272
in	10	196.2	205.5	1.288	1.260
SW	100	200.5	207.3	1.294	1.284

(b)	Semi-Space	Mark-Sweep	Gen-Copy	Gen-MS
antlr	0.20%	-0.05%	0.02%	-0.36%
bloat	-0.50%	1.09%	-0.28%	0.93%
fop	0.01%	0.24%	-0.07%	-0.58%
hsqldb	0.17%	0.32%	1.40%	-2.28%
jython	0.85%	0.95%	-0.62%	2.32%
ps	1.23%	-0.36%	-0.82%	-0.42%

for the garbage collector. Each measurement was performed ten times, resulting in 960 measurements total (6 benchmarks, 4 garbage collectors, 4 levels of HW overhead, 10 runs). The delay due to HW overhead is very small because even in these benchmarks, which are considered to be memory-intensive, the number of garbage collections is limited to values ranging from 7 (fop, SemiSpace) to 148 (hsqldb, GenCopy). We present only the numbers for the measurements with 25600 extra objects compared to those without HW. As can be seen from Table I(b), the introduced delay is never more than 2.32%. The non-determinism, introduced mainly by the optimizing compiler introduces noise in the measurements. This noise is the cause of the negative delays we measured.

VII. Conclusion

In this paper we have presented an interface between HW and the object-oriented high level programming language Java, which enables method calls over the HW/SW boundary while preserving the Java convention of passing objects by reference. We have adapted the garbage collection mechanism in Jikes RVM in such a way that four major garbage collectors of the Jikes RVM are supported. Measurements show that for a reasonable amount of references in HW (2560 object references), the absolute cost of garbage collection is low. The impact of garbage collector on memory-intensive benchmarks is limited to 2.32%, even for a large amount of object references in HW (25600 references).

In the future we plan to improve the garbage collection performance for generational collectors. We also plan to accelerate real-life application using the Java/HW interface and garbage collector we presented here.

Acknowledgements

This research is supported by grant 020174 of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), by GOA project 12.51B.02 of

Ghent University, by grant G.0021.03 of the Fund for Scientific Research – Flanders and by hardware donations from Altera. Philippe Faes and Dries Buytaert are supported by a scholarship of the IWT-Vlaanderen.

REFERENCES

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363– 1375, November 2004.

[2] Y. Ha, P. Schaumont, S. Vernalde, M. Engels, and R. Lauwereins, "A SW/HW interface API for Java/FPGA co-designed applets," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 269–270, april 2001.

[3] J.-Y. Mignolet, V. Nollet, P. Coene, D.Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of Design, Automation and Test in Europe (DATE)*, pp. 986–991, Apr. 2003.

[4] S. Trimberger, "Scheduling designs in a time-multiplexed FPGA," in *International Symposium on FPGAs*, pp. 153–160, ACM, 1998.

[5] Ph. Faes, M. Christiaens, and D. Stroobandt, "Transparent communication between Java and reconfigurable hardware," in *Proceedings of the 16th IASTED International Conference Parallel and Distributed Computing and Systems* (T. Gonzalez, ed.), (Cambridge, MA, USA), pp. 380–385, ACTA Press, 11 2004.

[6] K. Arnold and J. Gosling, *The Java Programming Language*. Addison Wesley, 1996.

[7] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *Reconfigurable Architectures Workshop (RAW)*, Apr. 2003.

[8] CoWare webpage, "CoWare, inc.." http://www.CoWare.com.

[9] M. Budiu, M. Mishra, A. R. Bharambe, and S. C. Goldstein, "Peer-to-peer hardware-software interfaces for reconfigurable fabrics," in *IEEE Symposium on Field-Programmable Custom Computing Machines*.

[10] DaCapo webpage, "The DaCapo benchmark suite." http://osl-www.cs.umass.edu/DaCapo/gcbm.html.

[11] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? high performance garbage collection in Java with MMTk," in *ICSE 2004, 26th International Conference on Software Engineering Edinburgh, Scotland, United Kingdom*, pp. 137–146, 5 2004.

[13] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss, "Beltway: Getting around garbage collection gridlock," in *Proceedings of PLDI'02 Programming Language Design and Implementation* (L. J. Hendren, ed.), (Berlin), pp. 153 – 164, ACM Press, June 2002.