

---

## Improving Java performance using dynamic method migration on FPGAs

---

Emanuele Lattanzi

ISTI – Information Science and Technology Institute,  
University of Urbino, piazza della Repubblica, 13, 61029 Urbino, Italy  
E-mail: lattanzi@sti.uniurb.it

Aman Gayasen, Mahmut Kandemir and  
N. Vijaykrishnan\*

CSE – Department of Computer Science and Engineering,  
Penn State University, 342 IST Building, University Park, PA 16802, USA  
E-mail: gayasen@cse.psu.edu E-mail: kandemir@cse.psu.edu  
E-mail: vijay@cse.psu.edu  
\*Corresponding author

Luca Benini

DEIS – Department of Electronics, Computer Science and Systems,  
University of Bologna, Viale Risorgimento, 2, 40136 Bologna, Italy  
E-mail: lbenini@deis.unibo.it

Alessandro Bogliolo

ISTI – Information Science and Technology Institute,  
University of Urbino, piazza della Repubblica, 13, 61029 Urbino, Italy  
E-mail: bogliolo@sti.uniurb.it

**Abstract:** With the diffusion of Java in advanced multimedia mobile devices, there is a growing need for speeding up the execution of Java bytecode beyond the limits of traditional interpreters and just-in-time compilers. In this area, Java coprocessors are viewed as a promising technology, which marries the flexibility of a general-purpose microprocessor to run legacy code and lightweight Java methods, with the high performance of a specialised execution engine on speed-critical bytecode. This work proposes and analyses a microprocessor with FPGA coprocessor architecture with efficient shared-memory communication support. Furthermore, we describe a complete run-time environment that supports dynamic migration of Java methods to the coprocessor, and we quantitatively analyse speedups achievable under a number of system configurations using an accurate complete-system simulator.

**Keywords:** FPGA; coprocessor; dynamic method migration; Java performance.

**Reference** to this paper should be made as follows: Lattanzi, E., Gayasen, A., Kandemir, M., Vijaykrishnan, N., Benini, L. and Bogliolo, A. (2005) 'Improving Java performance using dynamic method migration on FPGAs', *Int. J. Embedded Systems*, Vol. 1, Nos. 3/4, pp.228–236.

**Biographical notes:** Emanuele Lattanzi received the Laurea degree in 2001 from the University of Urbino, Italy. In 2001, he joined the Information Science and Technology Institute of the University of Urbino, as a PhD Student. In 2003, he was with the Department of Computer Science and Engineering of the Pennsylvania State University, PA, working as a visiting scholar with professor Vijaykrishnan Narayanan. His research interests are in the area of wireless embedded systems, with emphasis on power and performance analysis and optimisation.

Aman Gayasen received the BTech degree in Electrical Engineering from the Indian Institute of Technology, Delhi, in 2001. In 2002, he joined the Computer Science and Engineering Department of the Pennsylvania State University where he is currently working towards a PhD degree. His research interests include reconfigurable devices and systems, nanotechnology and low-power VLSI.

Mahmut Kandemir has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests

are optimising compilers, I/O intensive applications and power-aware computing. He received the BSc and MSc degrees in Control and Computer Engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD from Syracuse University, Syracuse, New York in Electrical Engineering and Computer Science, in 1999. He is a member of the IEEE and the ACM.

N. Vijaykrishnan is an associate professor in the Computer Science and Engineering Department at the Pennsylvania State University. His research interests are in the areas of energy-aware reliable systems, embedded Java, nano/VLSI systems and computer architecture. He has received several awards including the IEEE CAS VLSI Transactions Best Paper Award in 2002, the ACM SIGDA outstanding new faculty award in 2000, Upsilon Pi Epsilon award for academic excellence in 1997, the IEEE Computer Society Richard E. Merwin Award in 1996 and the University of Madras first rank in computer science and engineering in 1993. He is a Coeditor-in-Chief of the *ACM Journal of Emerging Technologies in Computing* and an Associate Editor of the IEEE Transactions on VLSI.

Luca Benini received the BS Degree (summa cum laude) in Electrical Engineering from the University of Bologna, Italy, in 1991 and the MS and PhD Degrees in Electrical Engineering from Stanford University in 1994 and 1997, respectively. He is an Associate Professor in the Department of Electronics and Computer Science at the University of Bologna. He also holds visiting researcher positions at Stanford University and the Hewlett-Packard Laboratories, Palo Alto, CA. His research interests are in all aspects of computer-aided designing of digital circuits, with special emphasis on low-power applications and in the design of portable systems. On these topics, he has published more than 150 papers in international conferences and journals. He is coauthor of the book: *Dynamic Power management, Design Techniques and CAD Tools*, Kluwer 1998. He is a member of the technical program committee for several technical conferences, including the *Design Automation Conference*, the *International Symposium on Low Power Design* and the *International Symposium on Hardware-Software Codesign*.

Alessandro Bogliolo is the Director of the Information Science and Technology Institute (ISTI), University of Urbino, Italy. He received the Laurea degree in Electrical Engineering and the PhD degree in Electrical Engineering and Computer Science from the University of Bologna, Italy, in 1992 and 1998, respectively. From 1992 to 1999, he was with the Department of Electronics, Computer Science and Systems (DEIS), University of Bologna. In 1995 and 1996, he was visiting the Computer Systems Laboratory (CSL), Stanford University, Stanford, CA. From 1999 to 2002, he was Assistant Professor in the Department of Engineering of the University of Ferrara, Italy. In 2002, he joined the University of Urbino, Italy, as Associate Professor. His research interests include embedded low-power systems, dynamic power management, signal integrity and bioinformatics.

---

## 1 Introduction

The market for mobile devices and phones is continuing to increase at a rapid rate. For example, the hand-held mobile device market in the USA is currently increasing at an annual rate of 22%.<sup>1</sup> Owing to a compelling set of capabilities including ‘Write once, run anywhere’, graphics, networking and security, Java is being employed increasingly in this domain. In fact, a recent report projects that Java will be the dominant terminal platform in the wireless sector, being supported by over 450 million handsets in 2007, corresponding to 74% of all wireless phones that will ship that year (McAteer, 2002). A major hindrance in achieving this projected dominance is the persistent perception that Java is “too big and too slow and not real-time enough” (The Embedded Software Strategic Market Intelligence, 2002).

A major reason for this perception is owing to the use of interpreters to implement the Java virtual machine (JVM) in the embedded devices. While interpreters are adopted for their simplicity and small memory footprint requirements, they have a poor performance, as the execution of each

bytecode requires tens of processor cycles. In order to target the performance drawback in the constrained environments, several hardware accelerators have been proposed for executing Java bytecode. The Java accelerators range from extensions to the native processors to stand-alone coprocessors that run in parallel with a host CPU. Examples of these Java accelerators include aJile’s JemCore, Sun’s picoJava, ARM’s Jazelle, Aurora VLSI’s DeCaf, inSilicon’s JVXtreme, Nazomi’s JSTAR, Parthus Technologies’ MachStream and Zucotto Wireless’ Xpresso (see Narayanan et al., 2002 for an overview). These accelerators either support direct execution of Java bytecode completely eliminating the need for interpretation (e.g., the picoJava processor) or support the acceleration of interpretation as in the Nazomi JSTAR.

A problem with using a pure Java accelerator is its inability to support existing legacy code. Alternatively, accelerators can be used as coprocessors (in addition to a legacy processor) or a single processor can be operated in dual-mode such as ARM’s Jazelle where one of the modes support bytecode execution. Application-specific extensions of the instruction set of a RISC processor can be

implemented by inserting a reconfigurable hardware module within the pipeline of the processor (Campi et al., 2003; Vassiliadis et al., 2003; Sima et al., 2002). Alternatively, coprocessors can be used to speed up JIT compilation (Radhakrishnan, 2001).

In contrast to these approaches for Java acceleration, we focus on enhancing the speed of Java applications by executing computation-intensive code segments on a reconfigurable hardware and interpreting (or just-in-time compile and execute) the rest of the code. The mapping of a segment of Java bytecode in HW can be performed by automated design flows taking Java bytecode as behavioural input specification and providing the corresponding bit-stream for a target reconfigurable hardware device (Davis, 2002; Karlsson et al., 2001). Hardware synthesis provides higher performance than direct bytecode execution, since the underlying hardware can be customised to match the application flow.

Several architectures have been proposed that make use of reconfigurable hardware coprocessor that executes in parallel with a JVM running on a microprocessor core. The hardware coprocessor can be used to implement either single bytecodes (Kent and Serra, 2002) or entire methods/code segments (Fleischmann et al., 1999; Radhakrishn et al., 2000). At bytecode level, the fine-grained interaction between HW and SW raises communication issues that, in most practical cases, limit the effectiveness of the approach. Communication issues are less critical when working at method level, since the encapsulation provided by Java methods can be used to drive HW-SW partitioning. Fleischmann et al. proposed a system architecture that makes use of a reconfigurable device to dynamically implement computation-intensive methods for which configuration bit streams were pre-computed offline (Fleischmann et al., 1999). Despite the granularity of the HW-SW partitioning, the effectiveness of their approach is still limited by communication issues. In fact, entire data structures need to be transferred across a *Java Native Interface* (JNI). Moreover, the code of the Java application needs to be modified in order to exploit the coprocessor. A more advanced architecture was proposed by Radhakrishn et al. (2000), where communication overheads are reduced by using a shared-memory approach. A strategy for mapping CPU-intensive methods on a reconfigurable device was also proposed by Ha et al. (2002), achieving platform independence at the cost of modifying the Java source code to enable run-time automated generation of configuration bit-streams.

In this paper, we present a hardware-software reconfigurable platform conceived to speed-up the execution of a target Java application by dynamically migrating critical methods on a reconfigurable hardware device. As in Radhakrishn et al. (2000), a shared memory is used to support HW-SW interaction. In addition, the JVM is modified in order to make hardware mapping transparent to the application.

The JVM collects usage statistics and selects CPU intensive methods that may benefit from a hardware implementation. The selected methods are then mapped on a reconfigurable hardware device while the main processor continues the execution of the Java application. Dynamic method migration to the reconfigurable device entails hardware configuration (i.e., upload of the configuration bit stream in the (re)configurable device), communication and synchronisation between main processor and reconfigurable device. Configuration bit streams can be either pre-computed offline and stored in a non-volatile memory for a finite library of candidate methods or generated at run time by a HW compiler that can be, in turn, either a software or a hardware component.

In the next section, we describe the architecture of the reconfigurable system, and we address the practical issues related to dynamic method migration, with particular attention to the support provided by the modified JVM. In Section 3, we present a cycle-accurate full-system simulation of the proposed architecture, built on top of Virtutech's Simics (Magnusson et al., 2002), that provides support for a thorough exploration of both system configuration (processor speed and architecture, memory hierarchy, bus arbitration policy, coprocessor performance, etc.) and run-time policies for dynamic method migration.

We use the simulation environment to analyse the effects of the main system parameters on the speedup achieved by means of dynamic method migration. In order to speed up exploration, the Java bytecode of the method to be mapped in hardware is directly used as a functional specification for the hardware coprocessor, while its configuration and execution times are annotated from real experiments conducted on the target reconfigurable device. We used a Xilinx Virtex-II for this purpose.

The rest of the paper is organised as follows. In Section 2, we describe the proposed architecture, and we discuss the issues related to dynamic method migration, hardware-software interface and synchronisation. In Section 3, we describe the full-system simulation environment used to perform performance evaluation and design space exploration. In Section 4, we present simulation results. In Section 5, we draw conclusions.

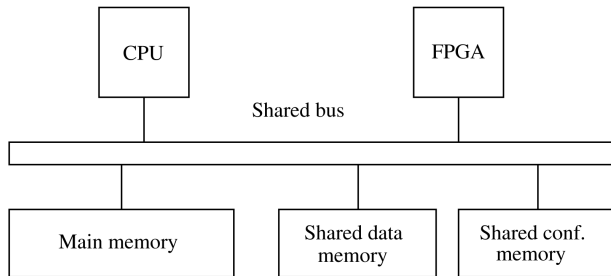
## 2 System architecture

The proposed architecture is shown in Figure 1. The key components are a processor (CPU) running the target Java application on top of a virtual run-time environment and a reconfigurable hardware device (FPGA) acting as a coprocessor. CPU and FPGA are connected to a system BUS that provides access to an asymmetric-memory system composed of three main components:

- the *main memory* used by the CPU to run system-level software, to build the Java run-time environment and to allocate software objects

- a *shared memory* used to allocate data that may be accessed both by the CPU and the FPGA
- a *configuration memory* used to store the configuration bit-streams for the FPGA.

**Figure 1** System architecture



All memory devices are mapped on the address space of the CPU, but they are used in different ways and may provide location-dependent performance. The main memory is accessed by the CPU through one or more levels of cache, while it cannot be accessed by the FPGA. The shared memory is viewed by the CPU as a non-cacheable memory, since it is mainly used for inter-processor communication. On the other hand, the shared memory is usually much smaller than the main memory and can be implemented by fast SRAM devices providing access times of a few clock cycles. Finally, the configuration memory is used to store configuration bit streams that can be either generated on the fly by means of a hardware compilation tool or pre-computed offline. In the first case, the configuration memory needs to be accessed at run time by the CPU to store the results of HW compilation. In the second case, the configuration memory is loaded offline and acts as a read-only memory during execution.

The FPGA is viewed by the CPU as a memory-mapped device, providing a set of interface registers that are made accessible by a specific device driver by means of *read* and *write* operations.

Since CPU and FPGA usually operate in parallel, bus arbitration mechanisms are required to handle bus contentions. Any bus architecture supporting multiple masters (such as AMBA peripheral bus, PCI bus, etc.) can be used for this purpose. Notice that using a shared bus may limit the actual speed-up provided by the coprocessor. Although more aggressive communication infrastructures may be used, we conduct our analysis for a shared bus in order to obtain performance estimates directly applicable to most embedded systems with minimal changes.

In the rest of this section, we describe the simple policy used to control dynamic method migration and the implementation details of the HW-SW support.

### 2.1 Dynamic method migration

Dynamic method migration is transparently controlled by the JVM that runs the main application, collects usage statistics about its methods, implements a dynamic policy for deciding which methods are to be mapped in hardware,

triggers hardware mapping and handles run-time switching between software and hardware implementations of a given code segment.

The key parameter used to drive hardware mapping is the *heat* of each method, which is obtained by counting over a sliding window the number of fetched bytecodes belonging to that method. The JVM keeps track of the heat of all methods by using an extra field added to their internal representation. Whenever a method is entered, its heat is incremented and compared with the maximum heat obtained so far. The hottest method (i.e., the method with highest heat) is the first candidate for hardware mapping. Both the heat profiler and the mapping policy are implemented within the interpreter loop of the JVM, corresponding to method invocation opcodes.

In order to be mapped in hardware, a method must satisfy several requirements. First, it has to be hardware-mappable, i.e., either a synthesiseable description or a pre-synthesised configuration file has to be available. Second, all the objects used by the method must be accessible from the reconfigurable device, i.e., allocated in shared memory. Third, there must be sufficient space in the reconfigurable device. Fourth, its heat must meet policy-dependent requirements.

Notice that the JVM keeps updating the heat of methods mapped in hardware in order to implement replacement policies. In particular, the coolest HW-mapped method (i.e., the method with the lowest heat among those mapped in hardware) is the best candidate for replacement.

Once a method has been selected for hardware mapping, the JVM enables the (re)configuration of the coprocessor. Although online hardware synthesis can be performed at this time, for the sake of simplicity, we refer to a library-based approach where configuration bit-streams are pre-computed and stored in the configuration memory for all hardware-mappable methods. In this case, reconfiguration consists of uploading the pre-computed bit stream from the configuration memory into the FPGA. For this purpose, the JVM provides to the reconfigurable device the base address and the size of the bit stream corresponding to the selected method, enables reconfiguration and keeps executing the Java application. Configuration is performed as a DMA transfer. The main processor keeps running during hardware configuration, and the method under mapping is executed in software until configuration is completed. Dynamic switching between software and hardware implementations of a given method is obtained by replacing the standard invocation opcodes (`invokeStatic`, `invokeVirtual`, `invokeSpecial`) with custom opcodes (`invokeStatic_FPGA`, `invokeVirtual_FPGA`, `invokeSpecial_FPGA`) in the run-time image of the Java bytecode. Notice that the same approach is used by most JVMs to implement execution speedup strategies. For instance, the Sun's Kilobyte Virtual Machine (KVM) ([java.sun.com](http://java.sun.com), 2003) uses custom opcodes (denoted by `_FAST` suffix) to invoke cached methods.

As soon as the JVM decides to map in HW a given method, its invocation opcode is replaced by the corresponding FPGA opcode. However, since the

configuration takes a finite amount of time, the method needs to be executed in software until configuration is completed. For this purpose, a Boolean flag initialised to 0 is associated with each HW-mapped method. The routine that executes the custom invocation bytecode checks the Boolean flag associated with the invoked method. If it is 0, the status of the FPGA is read from a memory-mapped register in order to check for configuration completion. If the configuration is still in progress, the Boolean flag is kept unchanged and the software method is invoked, otherwise, the flag is set to 1 and the execution of the method is committed to the hardware device.

## 2.2 Coprocessor interface

Communication between the main processor and the reconfigurable device is controlled by the JVM. In particular, whenever a custom invocation opcode is encountered, a specific routine is called that takes care of communication and synchronisation issues.

Interfacing a HW-mapped method with the JVM entails:

- granting access to shared objects
- passing input parameters
- returning output parameters.

In order to reduce communication overhead and to avoid data duplication, shared objects are allocated in the shared memory. For this purpose, the JVM needs to be modified in order to use two heaps: a *main heap* allocated in the main memory (local to the CPU) and a *shared heap* allocated in the shared memory. From the implementation stand point, the shared heap is created by a custom memory allocation routine (called `malloc_FPGA`) that takes as input the base address of the shared memory and returns a pointer to the shared heap. The actual allocation of an object is dynamically performed by the `mallocHeapObject` routine invoked by the interpreter of the new opcode. Shared objects are simply allocated by passing to the `mallocHeapObject` the pointer to the shared heap, rather than the pointer to the main heap, depending on the decision taken by the allocation policy.

We remark that objects allocated in the shared heap are not subject to garbage collection. Their de-allocation (i.e., replacement) is explicitly controlled by the memory-allocation policy.

Shared objects are made accessible to the FPGA by providing the pointers to their positions in the shared heap. This is done by the JVM by writing into a specific register of the reconfigurable device. Multiple objects require multiple write instructions. To avoid synchronisation problems, pointers are buffered by the FPGA and then used during execution to access the actual objects.

Both input and output parameters are treated as shared objects, in the sense that they are allocated in the shared heap and their pointers are passed to the FPGA. Input parameters of primitive types (e.g., integer, double, etc.) are the only exception, since they are not allocated in the heap, but directly passed to the FPGA by writing in specific

memory-mapped registers made accessible by the device-driver.

## 2.3 Synchronisation

The synchronisation between the JVM running on the main processor and the reconfigurable device implementing a HW-mapped method is based on the mutually-exclusive access to the shared memory.

Once all input parameters and object pointers have been provided to the FPGA, the JVM grants control of the shared memory to the FPGA and enables hardware computation. This is done by setting a Boolean flag of the shared memory that is checked by the MMU before granting access to the device. If the CPU tries to access the shared memory when it is granted to the FPGA, the CPU is stalled.

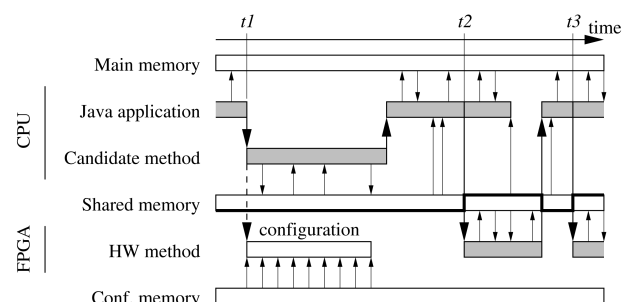
As soon as the FPGA completes execution it writes the results in the shared heap and resets the memory flag to return the control of the shared memory to the CPU. At this time, the stalled memory access is completed and the CPU resumes execution.

Notice that the CPU and the FPGA can execute in parallel both single-threaded and multi-threaded applications. In fact, invocations of HW-mapped methods are always non-blocking: once the execution of a HW-mapped method has been committed to the FPGA, the CPU keeps executing in parallel until it needs to access the shared memory (e.g., to get the results back from the FPGA).

On the other hand, resource contention may limit the exploitation of thread-level parallelism. This happens when the objects used by different threads (say, A and B) are simultaneously allocated in the shared memory. In this case, if the shared memory has been granted to the FPGA by thread A, thread B can execute only until it needs to access the shared memory. This performance drawback can be reduced by dynamically allocating into the shared memory only the objects that are needed by the HW-mapped method.

Synchronisation between hardware and software Java execution is schematically represented in the timing diagram of Figure 2. Both configuration and execution phases are represented. The difference between time intervals,  $[t1, t2]$  and  $[t2, t3]$  represents the speedup achieved by hardware mapping.

**Figure 2** Timing diagram of the interaction between a Java application running on the CPU and the FPGA implementing a Java method. Bold arrows denote method calls and control signals, while thin arrows denote memory transfers. A thick line is used to denote memory locks



This simple memory-lock mechanism has the key advantage of avoiding the overhead caused by the execution of the software routines needed to implement more advanced synchronisation strategies, based either on polling or on HW-interrupts.

### 3 Simulation-based exploration

We built a full-system simulation environment on top of Virtutech Simics, a system-level instruction-set simulator providing accurate control of the hardware architecture and visibility of run-time statistics and execution traces (Magnusson et al., 2002). Simics models a whole PC system (including interrupt controller, bus interfaces, disk, video memory, etc.) and provides a high degree of reconfigurability. In particular, it provides a programming language interface (PLI) to C/C++ that allows the users to create new hardware modules. We used the PLI to implement our reconfigurable device and shared-memory modules.

The shared BUS was implemented in Simics as a timing model (i.e., a module that implements the `timing_model_interface`) that has visibility of all memory operations and may stall the masters, thus providing support to the implementation of different arbitration policies. In particular, we used a simple TDMA protocol alternatively granting the BUS to the CPU and the FPGA for a fixed *time slot*.

We implemented an extra module to be used during our experiments to make the clock cycle count accessible from JVM. In this way we were able to easily measure the clock cycles taken to execute specific code segments.

We run Linux RedHat 6.2 (kernel 2.2.14) OS on our simulated platform with KVM 1.0.4 installed. KVM (Kilo Virtual Machine) is the Sun's virtual machine designed for resource-constrained environments (java.sun.com, 2003). It targets embedded computing devices with as little as a few kilobytes total memory. The KVM was modified as described in Section 2.

#### 3.1 Modelling the reconfigurable device

Modelling the reconfigurable device entails modelling both its functionality and its performance. The functional specification of the hardware device is directly provided by the bytecode of the methods mapped on it. In order to directly use such a specification, we encapsulated a *stack-oriented java processor* within the Simics module representing the reconfigurable device. This was done by porting the JVM interpreter source code on the Simics module.

Configuration time and HW performance were modelled by means of three parameters: *configuration-cycles-per-byte* that is the ratio between the total configuration time and the

size of the code segment; *execution-cycles-per-bytecode* fetched that expresses the effectiveness of HW mapping; and *shared-memory-latency* that is the access time of the shared memory.

The above-mentioned parameters are useful to decouple HW-SW design space exploration from HW-synthesis. In fact, they can be used either as sweep parameters during exploration or as constraints during HW mapping. On the other hand, they can be easily characterised for each mappable method by means of real experiments conducted on a target reconfigurable device. When configuration and execution times for all HW-mappable methods are taken from real experiments, system-level simulation provides cycle-accurate results.

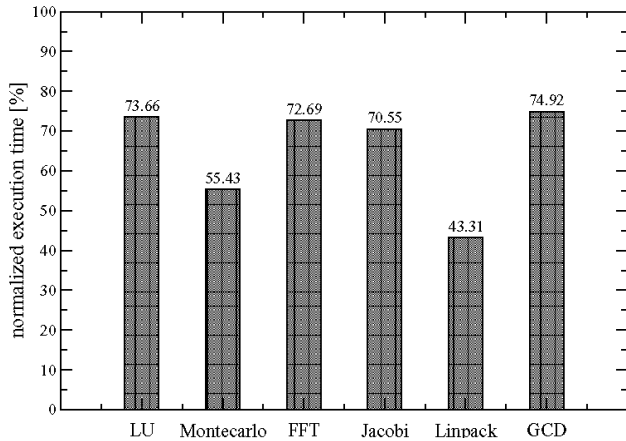
## 4 Experimental results

In this section, we report and discuss experimental results obtained by running some Java benchmarks on our simulated platform, with different configuration parameters. The simulated architecture was a low-power Embedded Pentium processor (running at 133 MHz, 266 MHz or 333 MHz), with 16 Kbyte of first level Dcache and Icache (with a miss penalty of 120 ns miss), 32 MB of RAM and 100 Kbyte of shared memory (with 40 ns access time). We characterised configuration and execution times for all HW-mappable methods by performing real experiments on a Xilinx Virtex-II FPGA.

We report two sets of experiments. First, we evaluate the speedup achieved by the proposed approach for fixed parameter settings. Second, we analyse the sensitivity to configuration parameters by using one of the benchmarks as a case study. The execution of each benchmark on a traditional platform without HW accelerator was also performed to provide baseline (*software-only*) performance values. All execution times are normalised to the time taken by the software-only execution of the same benchmark under the same system configuration.

#### 4.1 Speedup

The speedup achieved on each benchmark by means of dynamic HW mapping is shown in Figure 3 for a clock frequency of 133 MHz. All execution times are normalised to the time taken by the software-only execution. Results were obtained assuming conservative parameter settings: a configuration time of 100 cycles per byte, a HW-execution time of 1 cycle per bytecode, a shared memory access time of 10 clock cycles and a time slot of 1000 clock cycles for the BUS. The average speedup provided by the HW accelerator was 35%, with a maximum speedup of 45%.

**Figure 3** Benchmarks' execution time

We remark that we implemented a simple memory-allocation policy and a single-method mapping strategy, i.e., object allocation was decided by the JVM whenever a new opcode was fetched, and a single method at a time was mapped in HW. The observed behaviour of the HW-mapping policy was strongly dependent on the benchmark. In particular, a single method (out of 10 mappable methods) was mapped in HW when executing benchmark LU, 8 different methods (out of 15 mappable methods) were mapped in HW during the execution of benchmark Linpack.

The longest execution time was of about 16 million cycles, while the corresponding simulation time of Simics was of about five minutes on a 1.5 GHz Pentium 4 processor running Linux.

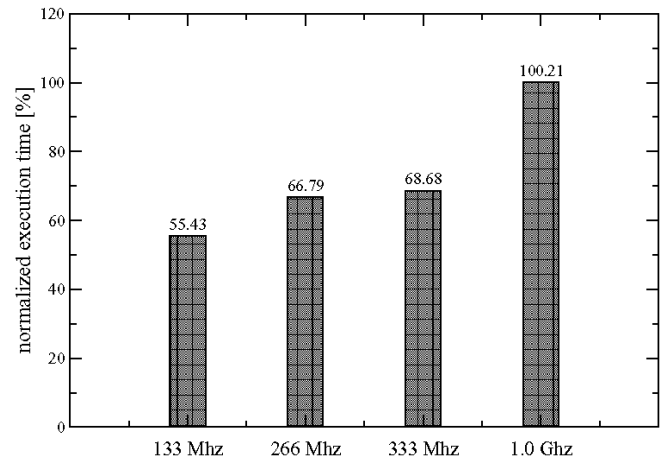
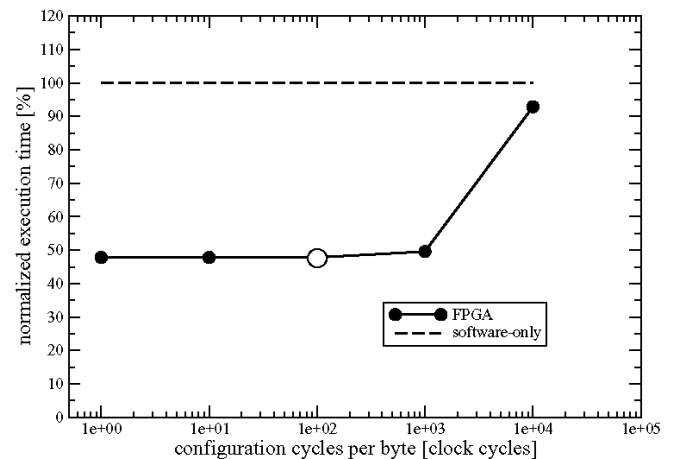
#### 4.2 Sensitivity analysis

In this subsection, we discuss the sensitivity to system parameters of the performance achieved by the proposed platform. Benchmark Montecarlo is used as a case study for this purpose.

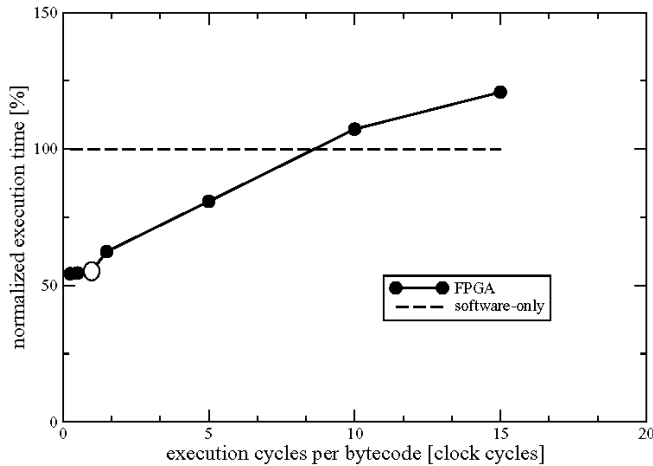
The first parameter that we analysed was the CPU clock frequency. We simulated the same benchmark at different clock frequencies while keeping unchanged all other parameters, including the absolute computation time of the HW coprocessor. As shown in Figure 4, the higher the CPU frequency, the lower the advantage of method migration. The break-even point for our case study was reached at 1 GHz which is an unrealistic speed for our embedded processor.

Figure 5 shows the dependence on the configuration time expressed in clock cycles per bytecode. The software-only execution time is reported in all graphs for comparison. Since it is independent of the parameters analysed in this section, it is always represented as an horizontal (constant) line. The parameter setting used in the previous subsection is denoted by an empty circle. The configuration time starts affecting performance only above 1000 clock cycles per byte. In fact, below this threshold, the configuration time is shorter than the time interval between subsequent invocations of the same method. Hence, the FPGA is

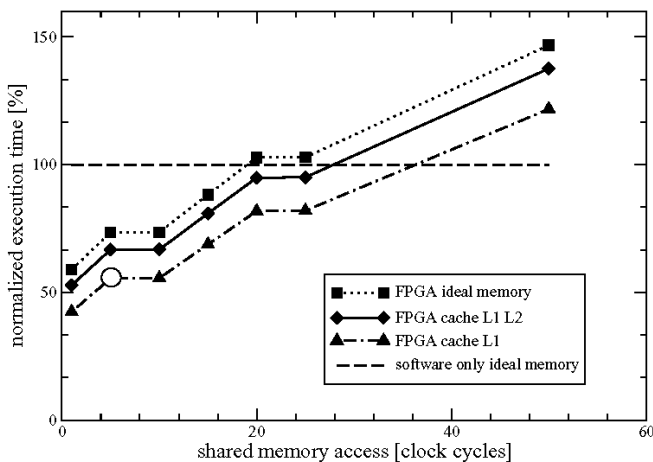
ready for execution the first time it is checked after HW mapping. Although this kind of behaviour is common to all benchmarks, the actual value of the threshold depends on the application. For our experimental setup, based on a Xilinx Virtex-II FPGA, we obtained an average number of *configuration-cycles-per-bytecode* of 100 clock cycles. We remark that this parameter strongly depends on the clock frequency of the main processor, the partial reconfiguration capability of the FPGA and the area on the mapped method.

**Figure 4** Changing CPU frequency**Figure 5** Sensitivity to configuration cycles per byte of method bytecode

The dependence on HW performance is shown in Figure 6. As expected, the overall performance is a linear function of the HW performance. It is worth noting that the break-even point (where HW mapping becomes counterproductive) corresponds to about 8 clock cycles per bytecode. This can be taken as a performance constraint for HW synthesis. On the other hand, HW-mapped methods are expected to provide much higher performance, taking less than 1 clock cycle per original bytecode [8]. For our benchmarks mapped on a Xilinx Virtex-II, we measured an average execution time of 0.2 clock cycles per bytecode fetched by the JVM to execute in software the same method.

**Figure 6** Sensitivity to HW execution cycles per bytecode

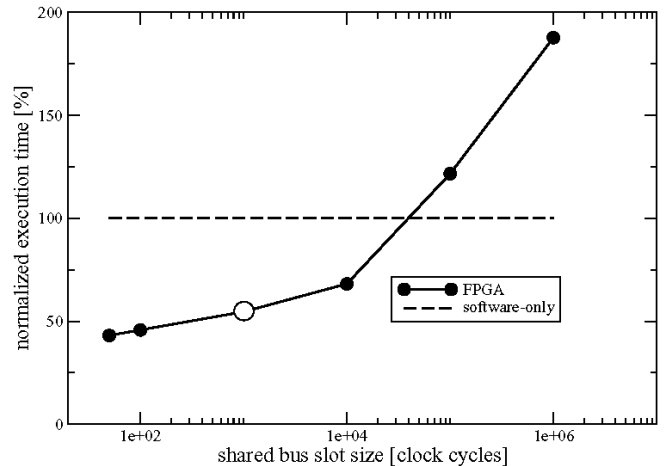
The effect of shared memory access time is reported in Figure 7. Since the observed effect depends on the performance gap between main memory and shared memory, the sensitivity analysis is performed for three different configurations of the main memory: (i) an ideal main memory, with 1-cycle access time (that is the worst case for our analysis, since a real shared memory is compared with an ideal main memory), (ii) the memory system described in Section 4 with first level Icache and Dcache and a hierarchical memory system with 16 KB L1 caches with 2-cycle miss penalty and a 512 kB unified L2 cache with 200ns miss penalty.

**Figure 7** Sensitivity to shared memory access time

For a fixed value of the shared memory access time, the better the performance of the main-memory system, the lower the advantage of HW mapping. The step-wise behaviour is owing to the interaction between the shared memory access time and the time slots of the shared bus. Notice that the performance metrics reported in Figure 7 are normalised to the performance of the software-only implementation. That is why a single (horizontal) curve is shown for comparison.

Depending on the configuration of the memory system, the break-even point that makes HW mapping counter-productive varies between 20 and 35 clock cycles.

Finally, Figure 8 shows the effect of the slot time used by TDMA BUS protocol. When the time slot exceeds 10,000 clock cycles, the overall performance is impaired because of the inefficient usage of the BUS.

**Figure 8** Sensitivity to bus-slot time

## 5 Conclusions

In this paper, we have proposed a coprocessor-based architecture for speeding up Java execution by means of dynamic method migration on FPGAs. We have devoted particular attention to reduce communication overheads, both through dedicated hardware support (shared memory and parallel configuration) and a modified Java run-time support system.

We developed a design-space exploration infrastructure, built on top of Simics, that enables full-system cycle-accurate simulation. We performed extensive experiments on a set of benchmarks to test the effectiveness of the proposed technique. The simulation infrastructure was used for this purpose, with HW-related parameters characterised by means of real measurements on Xilinx Virtex-II FPGAs. Experimental results, based on conservative assumptions, show that the proposed architecture consistently provides significant speedups: 35% on average by mapping in hardware only one method at a time.

## References

- Campi, F., Cappelli, A., Guerrieri, R., Lodi, A., Toma, M., La Rosa, A., Lavagno, L., Passerone, C. and Canegallo, R. (2003) *A Reconfigurable Architecture and Software Development Environment for Embedded Systems*, Reconf. Arch. Workshop.
- Davis, D. (2002) *Forge - High Performance Hardware from High-Level Software*, Xilinx technical paper ([http://www.xilinx.com/ise/advanced/forge\\_java.pdf](http://www.xilinx.com/ise/advanced/forge_java.pdf)).

- Fleischmann, J., Buchenrieder, K. and Kress, R. (1999) *Java Driven Codesign and Prototyping of Networked Embedded Systems*, Design Automation Conference (DAC).
- Ha, Y., Hipik, R., Vernalde, S., Verkest, D., Engels, M., Lauwereins, R. and Man, H.D. (2002) 'Adding hardware support to the hotspot virtual machine for domain specific applications', *Intl. Conf. on Field-Prog. Logic and Appl.*, pp.1135–1138.
- java.sun.com (2003) *Java 2 Platform Micro Edition (J2ME)*, <http://java.sun.com/j2me/>.
- Karlsson, D., Eles, P. and Peng, Z. (2001) 'A front end to a java based environment for the design of embedded systems', *4th IEEE Design and Diagnosis of Electronic Circuit and Systems (DDECS)*, April, pp.71–78.
- Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A. and Werner, B. (2002) 'Simics: a full system simulation platform', *IEEE Computer*, Vol. 35, No. 2, February, pp.50–58.
- McAteer, S. (2002) *Java will be the Dominant Handset Platform*, <http://www.microjava.com/articles/perspective/zeios>.
- Narayanan, V. and Wolczko, M.I. (2002) *Java Microarchitectures*, Kluwer Academic Publishers, Boston.
- Radhakrishnan, R., John, L.K., Vijaykrishnan, N. and Sivasubramanian, A. (2000) 'Architectural issues in java run-time systems', *International Conference on High Performance Computer Architecture*, January, pp.387–398.
- Radhakrishnan, R., Bhargava, R. and John, L.K. (2001) 'Improving java performance using hardware translation', *Intl. Conf. on Supercomputing*, pp.427–439.
- Kent, K. and Serra, M. (2002) *Hardware Architecture for Java in a Hardware/Software Co-Design of the Virtual Machine*, Euromicro Symposium on Digital System Design.
- Sima, M., Vassiliadis, S., Cotofana, S., Eijndhoven, J.T. and Vissers, K.A. (2002) 'Field-programmable custom computing machines: a taxonomy', *Intl. Conf. on Field-Prog. Logic and Appl.*, pp.79–88.
- The Embedded Software Strategic Market Intelligence (2002) *Java in Embedded Systems*, May, Vol. VII, <http://www.vdc-corp.com/embedded/reports/02/br02-31.html>
- Vassiliadis, S., Gaydadjiev, G., Bertels, K. and Panainte, E.M. (2003) 'The Molen programming paradigm', *Intl. Work. on Systems, Architectures, Modeling, and Simulation*, pp.1–7.

## Note

<sup>1</sup>US mobile devices to 2006: A land of opportunity, based on an extensive research program conducted by Datamonitor on mobile device markets in the USA.