Enabling Unrestricted Automated Synthesis of Portable Hardware Accelerators for Virtual Machines

Miljan Vuletić
Miljan.Vuletic@epfl.ch
Christophe Dubach
Christophe.Dubach@epfl.ch
Laura Pozzi
Laura.Pozzi@epfl.ch
Paolo Ienne
Paolo.Ienne@epfl.ch

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

ABSTRACT

The performance of virtual machines (e.g., *Java Virtual Machines*—JVMs) can be significantly improved when critical code sections (e.g., *Java bytecode* methods) are migrated from software to reconfigurable hardware. In contrast to the compile-once-run-anywhere concept of virtual machines, reconfigurable applications lack portability and transparent SW/HW interfacing: applicability of accelerated hardware solutions is often limited to a single platform. In this work, we apply a virtualisation layer that provides portable and seamless integration of hardware and software components to a JVM platform, making it capable of accelerating any Java bytecode method by using platform-independent hardware accelerators. The virtualisation layer not only improves portability of accelerated Java bytecode applications, but also supports runtime optimisations and enables unrestricted automated synthesis of arbitrary Java bytecode to hardware. To show the advantages and measure the limited overheads of our approach, we run several accelerated applications (handwritten and synthesised) on a real embedded platform. We also show our synthesis flow and discuss its advanced features fostered by the virtualisation layer.

Categories and Subject Descriptors: C.0 [General]: Hardware/software interfaces

General Terms: Design, Performance.

Keywords: Synthesis, Virtual Machine, Accelerator.

1. INTRODUCTION

Recent *Reconfigurable Systems-on-Chip* (RSoC) devices (e.g., Altera Excalibur, Xilinx Virtex-II Pro) offer to a potentially large number of users the possibility of coupling the CPU with arbitrary reconfigurable hardware accelerators. Although the use of reconfigurable hardware accelerators provides significant performance improvements [1, 9], a wider acceptance of the acceleration approach is hindered by nonstandard SW/HW interfacing and by the lack of cross-platform portability. The basic motivation for introducing virtual machines is achieving machine independence. However, reconfigurable applications—containing both software and reconfigurable hardware accelerators—inherently depend on underlying reconfigurable devices. In this work, we introduce an additional abstraction between software and hardware, and we employ it within virtual execution environments such as *Java Virtual Machines* (JVMs) to achieve portability and increase machine-independence.

Figure 1a shows a typical codesigned application built of software components (executed on a standard microprocessor) and hardware components (executed on reconfigurable logic such as *Field Programmable Gate Arrays*—FPGAs). Details of SW/HW interfacing are exposed to programmers and hardware designers. Besides recompiling software (written in a high-level programming language) and resynthesising hardware (described in a *Hardware Description Language*—HDL), porting to a different platform requires changing both hardware and software code. To support portability of reconfigurable applications, we introduce an additional abstraction layer (Virtual Interface in Figure 1b) that provides seamless SW/HW interfacing with limited performance penalties [14]. Despite the achieved portability, the application is machine dependent since its binary code and FPGA configuration bitstream are produced for a particular microprocessor and reconfigurable device.

A virtual execution environment—a virtual machine executing programs compiled to its virtual code—offers portability and machine independence on the software side (Figure 1c). Thanks to the virtualisation layer, the hardware side of applications is portable but not machine independent yet: one still has to write the code and map it eventually to the FPGA configuration bitstream. In an ultimate future system (Figure 1d), applications would be completely portable and machine independent. In the same manner as *Just-in-Time* (JiT) compilation (producing native ma-

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

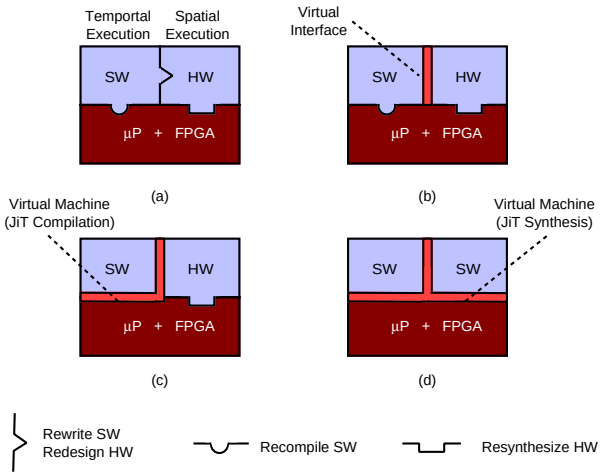

Figure 1: Portability and machine independence of reconfigurable applications.

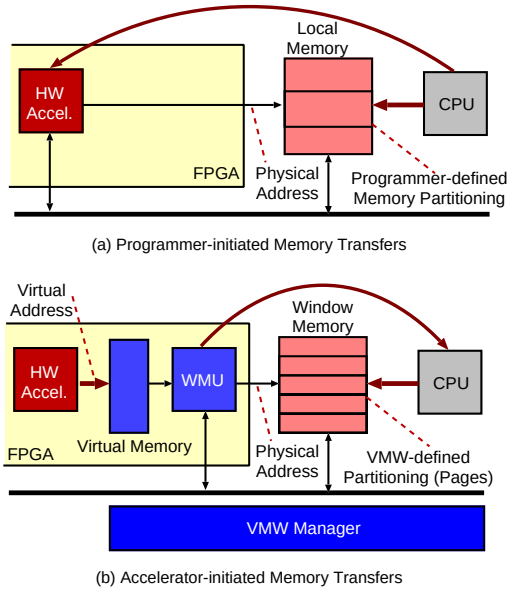

Figure 2: Handling memory transfers for hardware accelerators. In typical coprocessor cases (a), the programmer schedules memory transfers. In the presence of the virtualisation layer (b), the VMW manager performs memory transfers transparently to the programmer.

chine code from bytecode) is used nowadays, the future virtual machine should employ JiT synthesis (producing FPGA configuration from bytecode) to move *any* critical software section to reconfigurable hardware accelerators.

Apart from the challenge of synthesising hardware at runtime (runtime synthesis), the major difficulty here is to synthesise hardware without any restriction on the input bytecode (unrestricted synthesis): (1) in a general case, memory access behaviour of the bytecode is not known at synthesis time, thus, it is impossible to schedule memory transfers—servicing the accelerator accesses—in advance, and (2) not all of the bytecodes can be efficiently implemented in hardware. In addition, in present virtual machines without runtime synthesis (Figure 1c), increasing reusability of the generated HDL code requires portable hardware accelerators.

After related work in Section 2, we present an OS-based virtualisation layer that fosters portability and enables unrestricted synthesis in reconfigurable systems (Section 3). In Section 4, we integrate our virtualisation layer with a *Java Virtual Machine* (JVM), and we obtain portable hardware accelerators and create an environment for unrestricted automated synthesis (described in Section 5) of Java bytecode methods to hardware. Finally, we present our experiments and results in Section 6, and we conclude in Section 7.

2. RELATED WORK

Software [6] and hardware [8, 9] approaches have been used to accelerate code execution in virtual machines. In the case of JVMs, industrial solutions exist that support limited or complete subset of JVM instructions (Java bytecode) directly in hardware [9] (e.g., PicoJava from Sun, Jazelle from ARM, or DeCaf from Aurora VLSI). The approach improves execution times of Java programs, but it is highly dependent on the used JVM. Other authors have addressed the optimisation problem at a higher level, by implementing Java methods in reconfigurable hardware [5, 8]. However, these solutions are usually dependent on the underlying reconfigurable platform and the used JVM. In contrast to both approaches, our solution relies on a SW/HW virtualisation layer [14] that provides portability and seamless interfacing of reconfigurable accelerators (critical Java bytecode methods migrated to hardware). Our approach can be used with any JVM which conforms to the Java platform standard. Moreover, our extension to the JVM execution environment allows to run these accelerators without any modification of either compiled Java bytecode or Java source.

Hardware synthesis from high-level programming languages has been a challenging topic for a long time [11]. Typical challenges are to map high-level language concepts such as pointers, memory allocation, function calls, and recursion onto hardware. Memory allocation and management in embedded systems have attracted special attention of the researcher community [2, 10]. Some authors [15] use compile-time analysis to generate distributed memory structures with custom memory managers. On the contrary, our solution is a general-purpose one: memory management is performed by the OS in runtime and the virtual memory space is unique for all accelerators. This feature together with the callback makes possible synthesis of high-level programming language concepts such as memory allocation, function calls, and recursion [14].

3. SEAMLESS INTERFACING

Nonstandard programming paradigms and SW/HW interfacing models have certainly hindered the acceptance of reconfigurable computing. Our virtualisation layer (called *Virtual Memory Window*—VMW) addresses these problems [13, 14] by reusing the simple and well-known concept of virtual memory. The VMW allows hardware accelerators to share the virtual memory address space with user applications, thus simplifying the programming paradigm and hardware interfacing, and enabling unrestricted synthesis.

Dynamic handling of memory accesses. Figure 2 shows typical and virtualisation layer-based reconfigurable

(a)

```
/* Calling accelerator in a typical case */
int idea_cipher_coprocessor(int *A, int *B, int size) {
  data_chunk = DP_SIZE / 2; data_pt = 0;
  while (data_pt < size * 8) {
    memcpy(DP_BASE, A + data_pt, data_chunk);
    *IDEA_CTRL_REG = START;
    while(*IDEA_STATUS_REG != FINISH);
    *IDEA_STATUS_REG = INIT;
    memcpy(B + data_pt, DP_BASE + data_chunk, data_chunk);
    data_pt += data_chunk;
  }
}
```

(b)

```
/* Calling accelerator using virtualisation */
int idea_cipher_coprocessor(int *A, int *B, int size) {
  param.params_no = 3;
  param.flags = IDEA_CIPHER;
  param.p[0] = A;
  param.p[1] = B;
  param.p[2] = size;
  FPGA_EXECUTE(IDEA_HWACC, &param);
}
```

Figure 3: Invoking hardware accelerators for the IDEA cryptography application from C for a typical (a, burdensome) and virtualisation layer-based case (b, elegant). The system call FPGA_EXECUTE passes the parameters and transfers the execution to the accelerator.

accelerators running on behalf of the user application. In the typical case, the accelerator generates physical memory addresses of the local memory; the programmer has to partition the data (the size of the data to be processed does not necessarily fit the local memory) and schedule memory transfers from the main memory to the local memory, which requires knowledge about the accelerator memory access patterns (Figure 3a). In the virtualisation case, instead of being directly interfaced to the host platform, the hardware accelerator communicates to the rest of the system by using the virtualisation layer. The accelerator no longer generates physical memory addresses, but virtual ones translated by a hardware translation engine (called *Window Management Unit*—WMU, and functionally similar to the *Memory Management Unit*—MMU). The local memory is divided into pages and acts as a *Virtual Memory Window* (VMW). A part of the OS, the VMW *Manager* ensures that the translation is transparent to the end users. It provides a unified memory view to the main processor and the coprocessor. *Whatever* is the memory access pattern, neither programmers nor hardware designers *need to know anything about*. The virtualisation layer performs memory transfers (between the main memory and the window memory accessible by the accelerator) and enforces memory consistency dynamically at runtime, without any programmer intervention. Figure 3b shows an example of the transparent programming provided by the virtualisation layer.

Delegating tasks to SW. The virtualisation layer provides means for hardware accelerators to call back software. Figure 4 and Figure 5 illustrate how the virtualisation layer enables a hardware accelerator to call back software, even for sophisticated functions such as heap memory allocation (e.g., `malloc` from the standard C library; although it is usually not explicit in a virtual machine bytecode, virtual machines use a similar function for object creation). This feature further facilitates the synthesis and HW/SW partitioning. When resumed after the `malloc` finishes, the accelerator can use the pointer to allocated memory without any obstacles: the generated addresses will initiate the transfer of the accessed data to the local memory of the accelerator.

Portability. The virtualisation layer makes SW/HW interfacing seamless: calling a reconfigurable accelerator from

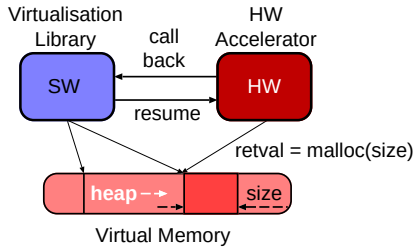

Figure 4: Memory allocation callback. Since hardware accelerators have unified memory image with software, using the *malloc* result is straightforward.

```
/* excerpt of a virtualisation library function */
...
struct cp_param param; /* parameter exchange structure */
...
FPGA_EXECUTE(HW_ACC, &params); /* coprocessor start */
...
while (!params.hwret) { /* wait for callback */
  switch(params.cback) { /* choose function to call */
    case 1:  ...  break;
    case 2:  ...  params.retval = malloc(params.p[0]); break;
    ...
    case n:
  }
}

FPGA_RESUME(HW_ACC, &params); /* coprocessor resume */
```

Figure 5: Library code for *malloc* callback. When the accelerator calls back the software, the library invokes the appropriate function and returns the virtual memory pointer back to the accelerator. Execution of the hardware accelerator is then resumed.

a user application is as simple as a common function call (e.g., the programmer can pass virtual memory pointers to accelerators). On the other side, designing (or synthesising from software) the accelerator hardware imposes no memory constraints (e.g., the hardware HDL code uses virtual memory addresses to access data). Application software and accelerator hardware are made portable: the VMW manager and the WMU hide platform-related details and make applications platform independent. Supposing that the virtualisation layer is already present, porting a reconfigurable application to a new platform would require just recompiling and resynthesising, *without any change* to the software and hardware source code (Figure 1b).

Dynamic optimisations. As it participates in application execution and communication transactions, the virtualisation layer can dynamically decide when and where to improve execution. For example, with additional hardware support in communication assistants, it can detect memory access patterns generated by hardware accelerators [12]. Then, it can predict future accesses and employ a prefetching technique trying to hide the memory copy latency.

4. VIRTUAL MACHINE INTEGRATION

We integrate the virtualisation layer within the JVM, for supporting migration of any Java bytecode method to hardware accelerators and for enabling their unrestricted automated synthesis. Using the JNI interface [7] is a typical and JVM-independent way to extend the JVM. The JNI defines, for a class, a native library (e.g., written in C) that implements some methods of the class. We use an extended class loader and a JNI-based native library to call hardware accelerators from the Java bytecode.

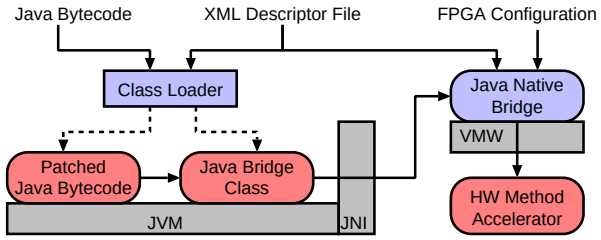

Figure 6: Steps involved in the execution of a coprocessor from Java. The class loader patches the application class and creates the bridge class. Calls to the accelerated methods are transfered to reconfigurable hardware through the JNI mechanism and our virtualisation layer.

```
/* IDEAEngine Java class */
void encrypt(
  byte[] data_in,
  byte[] data_out,
  int[] key
)
```

```
/* Bridge Java class */
void VMWrun(
  int methodId,
  byte[] data_in,
  byte[] data_out,
  int[] key
)
```

```
/* Bridge native library */
void VMWrun(
  int methodId,
  ...)  {
  FPGA_EXECUTE(&param);
}
```

Figure 7: Invoking hardware accelerators from a Java application. The original method is replaced with the call to a bridge class. The bridge class calls the native library which runs the accelerator.

Figure 6 shows the whole path from Java to coprocessor execution. An XML descriptor file and FPGA bitstream configuration accompany the Java bytecode of the application. The XML file defines which methods are to be executed in hardware and where the corresponding FPGA configurations reside. At class-invocation time, the class loader changes the Java bytecode of the application by replacing accelerated methods with calls to the corresponding native methods in the Java bridge class. Calling a native method, at execution time, invokes its native implementation through the JNI interface. The native implementation launches the hardware accelerator through the virtualisation layer. The described approach (1) is completely invisible for the programmer, (2) does not require changes in the original Java bytecode, and (3) does not depend on the used JVM.

Java Bridge Class. Our class loader constructs the Java bridge class dynamically at class-invocation time. It adds to the bridge class a native method prototype for each method to be accelerated in hardware. We use a predefined name (*VMWrun* in Figure 7) for all accelerated methods and we rely on the JVM capability to handle overloaded methods (same name but different number and type of arguments).

Java Native Bridge. The JVM links through JNI any native method of the bridge class to the same function in the native library. The application programmer is *unaware* of the existence of native methods and hardware accelerators. Its thread-based implementation is shown in Figure 8.

Parameter passing. The native implementation must handle different coprocessors: neither their parameter type nor their number is known at compile time. After using the method identifier to read the number of parameters and their types from the XML descriptor file, it retrieves the parameters thanks to the *variable arguments* feature of C.

Accessing JVM objects. The native implementation has to interpret correctly the type of parameters received

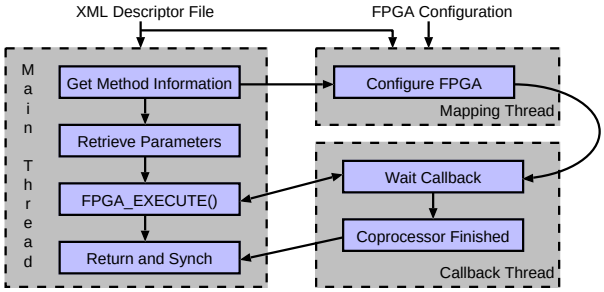

Figure 8: The three different threads of the native implementation. The main thread that contains the "entry point" and controls the accelerator through the VMW services, the mapping thread that configures the FPGA (for the first use) with the bitstream file, and the callback thread which services possible hardware callbacks.

from the JVM. If necessary, conversions are done using an appropriate JNI function. For example, obtaining a native pointer for an object reference—there are no pointers in Java—requires calling the JNI function. This discipline also maintains memory consistency, since the garbage collector does not move the acquired objects until their release.

Coprocessor Callback. To perform a callback (Section 3), an accelerator passes the method identifier with other arguments to the native function through the VMW interface (Figure 8). The callback feature allows performing high level operations (e.g., object allocation, synchronisation, virtual method invocation) that require support of the JVM. This is essential when it comes to unrestricted automated synthesis (Section 5) from Java bytecode to hardware.

5. UNRESTRICTED SYNTHESIS

As discussed in Section 3, the virtualisation layer facilitates unrestricted automated synthesis [4, 14]: sharing the same virtual memory, dynamically handling accelerator memory accesses, and having a standardised possibility to callback software enables synthesis of any Java bytecode method to reconfigurable hardware. Even sophisticated high-level language concepts such as object creation (recall the `malloc` example in Section 3), method invocation, and recursion can be executed in hardware with the help of the JVM. The presence of the virtualisation layer makes software to hardware migration easier: the migration could even happen dynamically, at runtime (as suggested in Figure 1d). Then, however, logic synthesis and place and route runtime become a major challenge—these issues are beyond the scope of this paper.

Figure 9 shows our basic synthesis flow (Compiler). Its inputs are Java bytecode of critical methods and a compiler configuration file. The flow consists of typical high-level synthesis phases [3]: (1) sequencing graph construction, (2) resource binding and operation scheduling, and (3) code construction. The generated VHDL code describes the interconnections between the resources and the finite state machines resulting from the scheduling step. An EDA synthesis tool and the FPGA vendor back end produce the bitstream file. Another output of the flow is the XML descriptor file—specifying which methods are mapped to the FPGA.

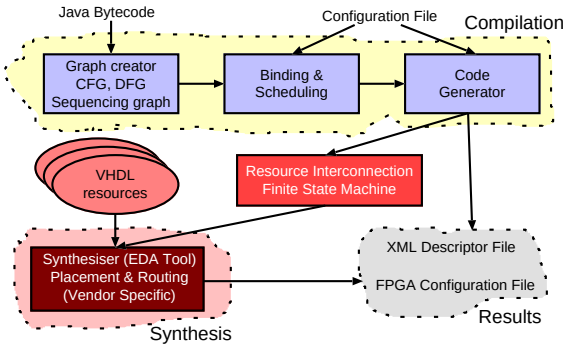

Figure 9: Automated unrestricted synthesis flow. Our flow creates HDL code of the accelerator and the XML descriptor file from the Java bytecode of methods to be accelerated.

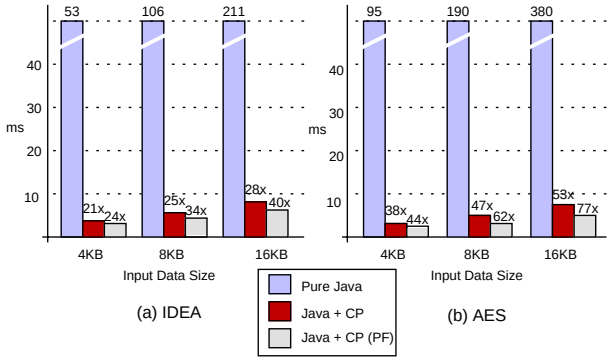

Figure 10: Execution times of the IDEA and AES applications for different input data sizes. Prefetching (PF) in the virtualisation layer improves performance: no user action is required.

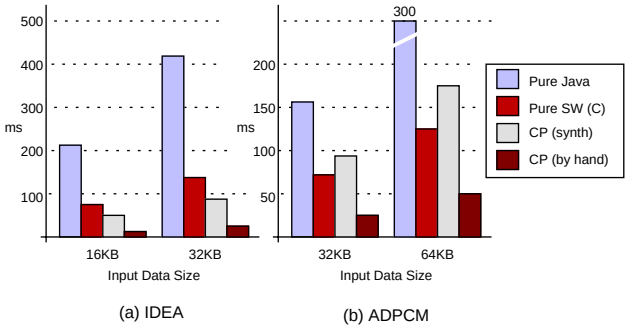

Figure 11: Execution times of synthesised IDEA and ADPCM coprocessors compared to Java, C, and hand-written coprocessors.

6. EXPERIMENTS

The measurement results refer to an embedded development board based on the Altera Excalibur device (containing an ARM processor running at 133MHz in the ASIC part, surrounded by the FPGA logic). Kaffe, an open source virtual machine, runs on top of a Linux kernel in the JiT mode. The virtualisation layer is implemented as a Linux module (the VMW manager) and a hardware component in the FPGA (the WMU). Although the FPGA we used is the smallest in its family (Altera EXPA1), the virtualisation area overhead is acceptable: the WMU uses no more than one-fifth of the FPGA resources. The time for FPGA configuration is not measured since it becomes irrelevant when the program is used for a longer period.

Speedup for Typical Input Data Sizes. First, in Figure 10, we present the speedup obtained for two handwritten cryptographic applications (IDEA—*International Data Encryption Algorithm*—in Figure 10a and AES—*Advanced Encryption Standard*—in Figure 10b), for typical input data sizes (for small data sizes the overhead is exposed, as we show in Figure 12). Significant performance improvements are achieved compared to the JVM-only execution of the Java bytecode applications, *without any explicit accelerator invocation in the application code*. The experiments in Figure 10 show that a dynamic optimisation in the virtualisation layer such as *prefetching* can be used transparently to increase performance. Beyond sequential memory accesses exercised by the two algorithms, future extensions of the virtualisation layer could also handle prefetching for other memory access patterns and minimise their impact on performance. Although the virtualisation layer introduces overheads, its presence can be transformed into an additional advantage: applications can benefit from runtime optimisations *without any change to the user software and hardware*.

Synthesis Results. Figure 11 shows execution times for two coprocessors synthesised directly from the critical Java bytecode methods of two Java applications (the IDEA cryptography and the ADPCM voice decoding application). The execution times are compared to pure Java bytecode executed in the JiT-enabled JVM, pure SW compiled to the machine binary code, and a handwritten VMW-based coprocessor. The current synthesis flow lacks some common optimisation passes to improve performance and exploit hardware parallelism. On the other side, relying on the virtualisation layer, it can map arbitrary Java bytecode to hardware. Although not as fast as handwritten, the synthesised coprocessors provide performance improvements, in comparison to the JVM with JiT. Furthermore, the performance improvement comes for free since there is *neither programmer nor designer intervention*: the synthesis flow produces the FPGA configuration and the XML descriptor file automatically (as shown in Figure 9).

The complexity of the synthesised coprocessors is comparable to the complexity of the handwritten ones. For example, the synthesised IDEA coprocessor occupies slightly more reconfigurable logic than the handwritten IDEA (4100 logic cells compared to 3600). Nevertheless, the logic in the handwritten coprocessor is used more efficiently, as the performance figures indicate. Improving the logic efficiency should be addressed by additional compiler passes.

Overhead Measurement. In Figure 12, we show the overhead of invoking the IDEA coprocessor for small input data sizes (one IDEA block equals to 8 bytes): execution time of the pure Java version against the execution time of the accelerated version. The execution time of the accelerated version is composed of several time components. The first is the overhead introduced by the Java bridge and its native library (parsing the XML descriptor file and creat-

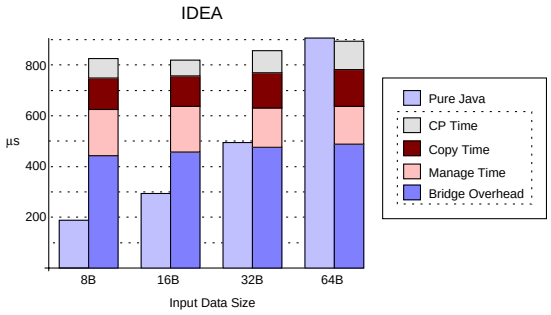

Figure 12: IDEA encryption of 8 bytes to 64 bytes (one to eight 64-bit blocks) of input data.

ing bridge class dynamically). Two thirds of the overhead are due to the configuration file reading; by optimising this operation, the overhead could potentially go below 200μs.

The other observable time components relate to the VMW interface. *Manage time* represents the time required by the OS module to manage the requests of the accelerator. *Copy time* is the time required to copy data (a whole page) from the main memory to the local memory of the accelerator. *CP time* is the time spent in the hardware execution, while the VMW interface is idle. The sum of *manage time* and *copy time* is constant, since for any data input size in the graph, only one page of memory is used (the page size is 2KB). Only the *CP time*, which is real computation time, is affected by input size.

For the two first data input sizes (8 bytes and 16 bytes), the time is almost the same. In fact this is due to the way that the IDEA coprocessor works: it has a three-stage pipeline that encrypts 3 blocks at a time; thus the time required to compute 1, 2, or 3 blocks is more or less the same. A difference for encryption of 8 blocks can be clearly seen (since the core computation is performed three times).

For 64 bytes (8 blocks), the coprocessor version becomes faster than pure Java software, despite the introduced overhead. If we recall that the overhead could be reduced (since the current implementation is not optimal), the break-even point can move toward even smaller datasets. What is more important, the total overhead (750μs) becomes negligible for typical input data sizes (Figure 10), especially when one bears in mind the benefits it brings: transparent and platform-independent acceleration of Java programs.

7. CONCLUSIONS

In this paper, we demonstrate how a virtualisation layer for SW/HW interfacing enables mapping *any virtual machine code* to hardware and supports portability of reconfigurable applications. As a proof-of-concept, we integrate the virtualisation layer within a JVM, allowing transparent use of hardware accelerators to programmers: *no change is required in the application software.* We also show a basic implementation of our unrestricted synthesis flow for JVM platforms. The measurements prove the viability of our approach: the introduced overhead is limited; dynamic optimisations in the virtualisation layer improve the performance *without any change of either Java bytecode or hardware accelerator.* Future work should employ typical synthesis optimisations and further exploit hardware parallelism.

8. REFERENCES

[1] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, Apr. 2000.

[2] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecapelle. *Custom Memory Management Methodology.* Kluwer Academic, Boston, Mass., 1998.

[3] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, New York, 1994.

[4] C. Dubach. Java Byte Code synthesis for reconfigurable computing platforms. Master thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), Jan. 2005.

[5] J. Fleischmann, K. Buchenrieder, and R. Kress. Java driven codesign and prototyping of networked embedded systems. In *Proceedings of the 36th Design Automation Conference*, pages 794–97, New Orleans, La., June 1999.

[6] The Java hotspot virtual machine (white paper). `http://java.sun.com/products/hotspot/`, 2002. Sun Microsystems.

[7] Java Native Interface specification. `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`, 2003. Sun Microsystems.

[8] E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo. Improving Java performance by dynamic method migration on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop*, Santa Fe, N. Mex., Apr. 2004.

[9] M. Levy. Java to go: The finale. *Microprocessor Report*, 4 June 2001.

[10] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip.* Kluwer Academic, Boston, Mass., 1999.

[11] L. Séméria, K. Sato, and G. De Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-9(6):743–56, Dec. 2001.

[12] M. Vuletić, L. Pozzi, and P. Ienne. Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors. In *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, pages 596–605, Antwerp, Belgium, Aug. 2004.

[13] M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Proceedings of the 41st Design Automation Conference*, pages 948–53, San Diego, Calif., June 2004.

[14] M. Vuletić, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers*, 22(2):102–13, Mar.–Apr. 2005.

[15] S. Wuytack, J. L. da Silva Jr., F. Catthoor, G. de Jong, and C. Ykman-Couvreur. Memory management for embedded network applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-18(5):533–44, May 1999.