Automatic Generation of Application–Specific Systems Based on a Micro-programmed Java Core

F. Gruian, P. Andersson, K. Kuchcinski,
Dept. of Computer Science, Lund University
Box 118, S-221 00 Lund
Sweden
<flagr, pera, kris>@cs.lth.se

M. Schoeberl
JOP.Design
Strausseng. 2-10/2/55, A-1050 Vienna
Austria
martin@jopdesign.com

ABSTRACT

This paper describes a co-design based approach for automatic generation of application specific systems, suitable for FPGA-centric embedded applications. The approach augments a processor core with hardware accelerators extracted automatically from a high-level specification (Java) of the application, to obtain a custom system, optimised for the target application. We advocate herein the use of a micro-programmed core as the basis for system generation in order to hide the hardware access operations in the micro-code, while conserving the core data-path (and clock frequency). To prove the feasibility of our approach, we also present an implementation based on a modified version of the Java Optimized Processor soft core on a Xilinx Virtex-II FPGA.

Categories and Subject Descriptors

C.3 [**Special Systems and Application-Based Systems**]: real-time and embedded systems

Keywords

system-on-chip, co-design, FPGA, Java

1. INTRODUCTION

It is well known that hardware accelerators can give both speed-up and reduced power consumption. An attractive technology for implementing accelerators is reconfigurable logic. Today it is even possible to build entire systems on a single FPGA, leading to a dramatic decrease in design time and cost compared to ASIC based systems, making them increasingly useful in embedded systems. Single-chip systems can be more tightly coupled, since the interconnects are no longer limited by the pad amount, as in multi-chip solutions.

The features just mentioned open new possibilities for application-specific systems. Hardware accelerators and co-processors can now be more deeply integrated with processor cores. Furthermore, accelerators may be automatically extracted from the software source code, grouped and merged together depending on the space available on the device. In addition, this process may also be made transparent to the user, who, unlike with other approaches for application-specific system design, involving application-specific processors and accelerators, does not have to be aware of the underlying hardware, nor provide specific compilers or libraries for accessing the accelerators.

In this paper, we propose a co-design flow for generating application-specific systems-on-chip, hosted on a FPGA. The target architecture consists of a processor enhanced with one or several accelerators. We also describe and discuss trade-offs at different steps in the design flow. Specifically we look at interconnection structures, communication and synchronisation issues with respect to both the accelerators and the modifications to the processor. In addition, we illustrate the proposed design flow using our version of the Java Optimized Processor (JOP, [10]), which is augmented with accelerators. The accelerators are automatically extracted form the application Java source code, the same code that can be run on JOP without accelerators.

In traditional methodology, the use of hardware accelerators implied a considerable design effort. The function to be accelerated had to be identified and then re-implemented, by an expert, using a hardware description language. To reduce the design overhead for using FPGA accelerators, much work has been done to automate this process. In [4], Gokhale et al. present the NAPA C compiler for the NAPA1000, a hybrid RISC/FPGA processor. They extend ANSI C with *#pragmas*, which steer whether variables and expressions are to be mapped onto the RISC, or the FPGA. Their hardware generation is however limited to scalar expressions (no memory accesses, conditions, or nested loops).

An approach for automatic accelerator generation in a Java environment is presented in [2]. It uses the Java bytecode as input for the accelerator generation. Data path generation is based on predefined and pre-synthesized macros, and only the generated controllers need to pass through logic synthesis. In contrast we generate RTL-level VHDL code for both the data path and controllers, starting from source code. We believe this is more flexible and increase the possibilities for optimizations. In [2], the Java Virtual Machine (JVM) and the accelerators do not share a common memory, so data are copied between the JVM memory and the accelerator local memory before and after the accelerated computation. This leads to a significant difference in the

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

generated interface compared to our approach, where the JVM and accelerator do share a common memory.

In [8] Kent et al. present a hardware/software co-design of a JVM, on a general purpose processor connected to an FPGA. Unlike, that approach which focuses on speeding up a JVM in general, for any Java application, our approach targets application-specific systems-on-chip by automatic extraction of accelerators at compile time.

There have been many more approaches to accelerator generation, most of them more or less automated. A survey of such systems and tools can be found in [3], for example.

Recently, much work has been done on the topic of compiling traditional software languages directly into hardware [1, 5, 11]. This makes a good foundation for fully automated accelerator generation directly from a software description.

The remainder of the paper is organized as follows. Section 2 gives an overview of our application specific system design flow. Section 3 presents the architectural options for implementing such systems, and motivates our choices. In section 4, we give the implementation details specific for our design flow and platform. Section 5 presents an experimental evaluation that validates our design methodology. The paper concludes with a summary in section 6.

2. CO-DESIGN FLOW OVERVIEW

Our view on the co-design flow for generating an application-specific system is detailed in Figure 1. The input to the whole process is a high-level specification of the application (in our case in Java), including all the necessary sources, preferably together with libraries and packages. The application is *profiled* in order to obtain essential data for selecting code regions suitable for hardware acceleration. Time consuming loops, frequently called short functions, and I/O operations are good candidates for hardware implementation. The application is then *partitioned* into software and hardware, taking into account a number of requirements such as performance, device utilization, micro-code memory size, etc. Although this step could be fully automated, we believe a tight interaction between the designer and the partitioning tool is beneficial at this point. For example, an experienced designer might directly recognize reusable code regions, only slightly different, that could share the same hardware. At this point, the application is described by a software part, further handled by the software flow, and a number of hardware functions (still described in Java in our implementation) that will undergo the hardware flow.

2.1 Software Flow

After partitioning, the software part includes, besides the actual code (bytecode), also specific hardware calls that replace the code to be implemented by the hardware accelerators. In fact hardware calls are new instructions, implemented at the micro-program level. To make the software flow more generic and keep a unique micro-code for all applications, we adopted in principle three new instructions (hardware calls): **hwWrite** to write a word to a certain hardware accelerator, **hwSynch** to synchronize (wait for results) with a certain accelerator, and finally **hwRead** to read a word from a given accelerator. In our implementation, these hardware calls are specific functions, replaced by dedicated bytecodes using a custom JavaCodeCompactor (JCC). Otherwise the software flow involves the classic compilation and static linking steps, to get the executable image.

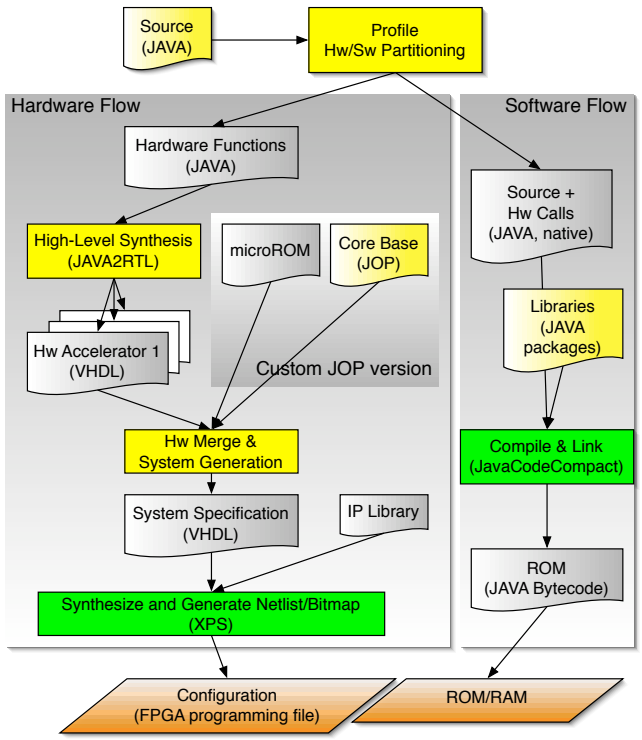

Figure 1: Our co-design flow for generating application–specific systems.

2.2 Hardware Flow

Once the hardware functions are decided, each of the corresponding Java code regions will be translated into a register transfer level specification RTL (a VHDL component in our case).

Our translator supports the semantic overlap of Java and C, i.e. loops, arrays and object references. We do not support yet recursion, inheritance, or the dynamic aspects of Java, i.e. object creation or dynamic class loading. The focus of our translator is on the memory accesses. We analyse the memory accesses looking for data reuse and patterns, allowing memory accesses trough bus bursts. To exploit data reuse we introduce local memories, drastically reducing the number of accesses over the shared bus [1]. The translator is intended for small loops and therefore we do not consider resource sharing. However, if large portions of code are to be translated, high-level synthesis employing resource sharing should be considered [5]. The translated entities will be later on connected to the initial base core (in our case JOP) and possibly the system bus, according to one of the choices described in section 3. All the components are then glued together during a hardware merge and system generation step. The system based on the micro-programmed processor, including the hardware accelerators and necessary peripherals is the result of this process (specified as VHDL, micro-code memory contents, and possible third party IPs).

Finally, this specification is synthesized, producing a configuration file for the desired FPGA (at this point our flow makes use of the Xilinx Platform Studio, CoreGenerator, and Xilinx ISE).

3. ARCHITECTURAL CHOICES

The specific hardware functionality extracted during the partitioning step can be incorporated into the system in a number of different ways, each of these having certain advantages and drawbacks. Next we review briefly three architectural options, pointing out our choices and the motivation behind them.

A first, straightforward option is to simply connect all the hardware accelerators as *slaves* (or *master/slaves*) to the common system bus (see Fig. 2, *Pheripheral 1*). All accesses occur exclusively through the bus, with the exception perhaps of optional interrupt requests to the processor. Note that the accelerators have to be accessible through normal bus read/write operations. The advantage of such an architecture is that, in principle, any processor core can be accelerated this way. No special instructions for accessing the hardware registers are required. From the software part, hardware calls can be simply implemented as read/writes to the memory addresses associated with the accelerators. Furthermore, the accelerators can run in parallel with the processor. However, because of using the common system bus as a point of access, each read/write operation to a hardware register takes a rather long time and competes with other bus transfers. For these reasons, this architecture is suitable only for complex/coarse-grain hardware operations.

A second, more tightly coupled architecture implies the use of a local access structure and special micro-instructions for reading/writing hardware registers (see Fig. 2, *Hw1*). The advantages over the previous method are the following. The hardware accelerators are not mapped over common addresses and cannot be accessed from outside except through dedicated micro-code. Malicious or badly designed Java code cannot therefore directly affect the hardware accelerators. In addition, the read/write operations can be constructed to be very simple and fast, since they do not have to abide the system bus protocol. As a consequence, even the hardware may become less complex, since the bus interfaces are not necessary. These features make this architecture better suited for finer grain operations. However, if the accelerators must access data from the memory, they can only do this through the core fetch unit, using micro-code operations. For large amounts and constant flow of data, this becomes an issue. This problem can be simply solved by combining the two approaches just presented. In particular, the accelerators can be connected as *master*s on the system bus, fetching the required data themselves instead of using the core fetch unit (see Fig. 2, *Hw2*). This way the hardware accelerators are still unaccessible (unless through the dedicated hardware calls) from the Java code, yet they can freely use the (shared) memory and peripherals while the core carries out other tasks. In our implementation, we decided to use both the pure local access structure and the combined local–bus master access choice, as in Fig. 2.

Finally, a third possibility would be to integrate the hardware functions with the core execution unit. This would allow for the new hardware operations to access the core registers and also share functional units among themselves and the core. However, this approach (designing an Application-Specific Instruction-set Processor) would drastically alter the data-path of the processor. In addition, the micro-decoder unit would have to accommodate specific instructions for each hardware function. These changes would impact the processor clock frequency to an extent that could need a detailed examination and re-design by an experienced hardware designer. Furthermore, the parallelism existing in the other architectures is lost, as the execution stage can carry out only one operation at one time. Although this approach might be useful for very fine grained operations (such as the DSP *multiply-accumulate*), we considered it to be unsuitable for fast and automatic system generation.

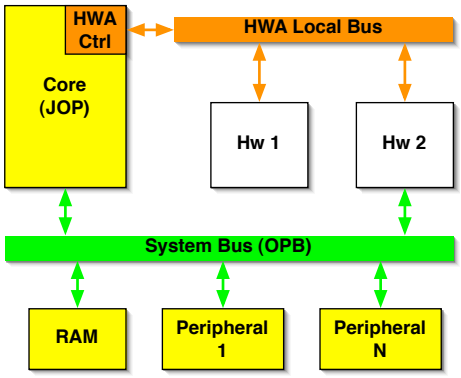

Figure 2: Our architectural choice.

4. IMPLEMENTATION DETAILS

The current section describes some of the implementation details specific to the design flow introduced above. In particular, we present the basic system organisation, the way of rewriting the Java code using hardware calls for the accelerated sections, the associated micro-code for these hardware calls, and the method of accessing the hardware accelerators. A brief description of some trade-offs and optimisation possibilities concludes this section.

4.1 Basic System Architecture

As support we use the MB1000 hardware platform [7], which is a Virtex-II evaluation board. The system used for evaluating our methodology is based on a Java Optimised Processor (JOP, [10]) core augmented with an On-chip Peripheral (OPB, [6]) *master* interface.

The JOP version used in our systems is a three stage pipeline, stack oriented architecture. The first stage fetches bytecodes from a method cache and translates them to addresses in the micro-program memory. The method cache is updated on invokes and returns from an external memory (through the system bus). The second stage fetches the right micro-instruction and executes micro-code branches. The third decodes micro-instructions, fetches necessary operands from the internal stack, and executes ALU operations, loads, stores and associated stack spills or fills. The internal stack consists both of a dual port RAM and two registers holding the TOS and TOS-1. Due to its organisation, JOP can execute certain Java bytecodes as single micro-instructions, while more the complex ones as sequences of micro-instructions or even Java methods.

In addition, our version of JOP uses memory mapped I/O, with all external access made synchronous (the processor is stalled until data becomes available) to accommodate different latencies for the different peripherals. Furthermore, the processor was extended with a Hardware Accelerators Local Bus (HWALB) consisting of 32 bit-wide address, data

in and out signals, plus a select and a read-not-write control signals. The HWALB protocol is trivial, each access having a latency of one clock cycle. Currently the accelerators are selected through both the global select and dedicated bits from the address bus, limiting[1] the number of connected accelerators to thirty-two.

Besides the JOP and the OPB themselves, the system contains a RAM (on-chip block RAM, or on-board memory), an UART (for loading applications), a Timer (the real-time clock), all connected to the OPB. However, any system configuration making use of the OPB is possible. The non-custom cores are those shipped with the Xilinx EDK 6.2.

4.2 Hardware Calls Structure

The decision of which parts to select for hardware implementation is currently taken exclusively by the designer. Once the code sections to be accelerated are selected, these are replaced by very specific function calls. Each such section of code is given an identifier, which will be the address of the hardware accelerator *hwID*. This identifier is one of the parameters to be passed on to the hardware calls[2]. The selected section is then replaced by a sequence of function calls as follows. The parameters required by the hardware are passed on using one **hwWrite(N, hwID)** for each parameter. On the hardware side, once all the necessary parameters are received, the hardware starts executing. If the accelerator is read via HWALB during execution, a non-zero number is output. When the execution is completed, the output becomes zero, followed by a sequence of results (see section 4.4). On the software side, the application has to execute a **hwSynch(hwID)** which is a busy wait for the *zero* (finished) value from accelerator. Finally, the results and/or modified variables are read back using **hwRead(hwID)**. At compile time JCC will identify the JOP micro-code calls, generate the right calling structure and use the dedicated bytecode for the appropriate hardware accelerators entry. Note that hardware calls can be interleaved in order to take advantage of the parallelism existing in the architecture.

4.3 Micro-instruction Set Extension

Besides the initial JOP micro-code, we added, as just mentioned, micro-code calls for writing parameters to the hardware, synchronising with the accelerators, and reading results from the hardware. We managed to write these general enough, such that no changes are required whenever hardware accelerators are added or removed. For this purpose, the micro-instruction set was extended to include:

- **stma** – a select hardware micro-instruction, which outputs the top of the stack on the address bus of HWALB. This is not a new micro-instruction, however, since it is also used for selecting the OPB address.
- **sthr** – a store hardware, which writes the top-of-stack to the hardware selected by a *stma*, causing the accelerator registers to shift values (see section 4.4).
- **ldhr** – a load hardware, which reads a word from the hardware selected by a previous *stma*.

[1]Slightly more complex address decoding schemes, supporting even 2^{32} accelerators can be implemented without great impact on performance and/or device utilization.

[2]The alternative of having a dedicated bytecode for each accelerated section might limit the number of hardware accelerators more drastically.

As opposed to the OPB read/write instructions for accessing the memory and peripherals, the hardware read/writes are non-blocking, assuming that the hardware can read data or provide results in a single clock cycle. An architecture without a HWALB could use the OPB bus and memory mapped hardware accelerators, without modifying the micro-code, however with the drawbacks mentioned in section 3.

4.4 Accelerator Access Structure

The description given until now covered mainly the processor side, however there are a few choices made on the accelerator side as well, leading to the structure depicted in Fig. 3. In particular, the registers holding the input values for the accelerator are chained, making up a FIFO, accessed through the same address. Each *sthr* micro-instruction basically pops the JOP top-of-stack and pushes it into the accelerator FIFO. A similar structure was adopted for the output registers. However, the output is dependent also on the accelerator state. A non-zero value is output as long as the results are not available (i.e. during computation). A *zero* is output as soon as the results are ready for reading, followed by a new output value each *ldhr*. Synchronisation between JOP and an accelerator is thus solved by polling the accelerator output until a zero is read. It is important to stress here that the input and output registers are not dedicated resources, but can be used during computation, and in most cases a single register is used as both input and output, representing one variable from the Java code.

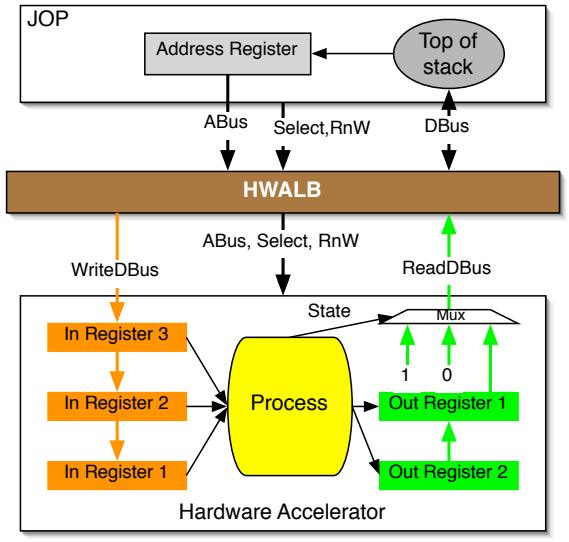

Figure 3: Accelerator access structure detail.

Other access structures are of course possible, such as assigning each hardware register to a different HWALB address or using interrupt driven synchronization. However, we decided that giving individual addresses to each registers would complicate the structure (another decoder), without any real gain, since all the parameters must be passed on anyway every time the accelerator is used. As for the interrupt driven synchronization, it is still under investigation.

4.5 Trade-offs and Optimizations

While going through our design flow, we were confronted

with a number of trade-offs and possibilities for optimisations that we wish to point out next.

The first trade-off is the actual hardware/software partitioning step, in which speed is traded for device utilisation. This is obviously greatly dependent on the application and for now strongly controlled by the designer, which could however benefit from a estimator/profiler especially built for our hardware/software architecture.

Deciding the actual structure of a hardware call is another interesting problem. Initially we decided to implement micro-code calls that would pass a variable number of values to the hardware, using an array of integers. The micro-code needed for this was rather complex, and required us to initialise an array for each hardware call. The call overhead as such was small, since only two parameters (the hardware address and the parameter array address) had to be passed on, while looping through the array was done at micro-code level. However, preparing the hardware call by transferring the parameters into the parameter array was a hidden overhead (i.e. array initialisation). We then decided to implement instead micro-code calls for passing a single parameter at a time. The micro-code directly became much simpler (no loops) and no additional arrays were needed. Furthermore, the number of parameter exchanged with the hardware seemed to be rather small (maximum eight) in our examples. Finally we intend to implement specialised calls for one, two, three, and four parameters, in order to reduce the size of the Java bytecode.

Another issue laid in selecting the most efficient architecture/interconnect structure for the hardware accelerators. There are advantages and drawbacks both with system bus accessed accelerators and with locally accessed hardware units, depending on the amount and localisation of data required by the accelerator. A combination between HWALB-slave and OPB-master seems to be the best in our case.

Beside the issues just mentioned, there are a number of trade-offs that could be investigated. For example, allowing some of the hardware accelerators to merge, would result in reduced device utilisation, but possible in increased computation latency. To improve merging, one could examine the Java source, to detect hardware functions that are never executed concurrently. In addition, using specific micro-programs for each accelerator might increase the performance at the expense of the micro-code memory size, yet another optimization problem.

5. EXPERIMENTAL EVALUATION

To examine the feasibility and efficiency of our design flow, we chose a simple application and run it through the design process. The application consists of an element-wise addition of two vectors of integers, followed by a summation of all the elements in the result vector, and a conversion from integer to a hexadecimal ASCII representation of the final sum. Although our goal is an almost fully automated flow, some of the steps were carried out manually, as not all the tools were yet available. However, these steps were relatively few, namely selecting the code to be implemented in hardware, replacing these code sections with proper hardware calls, and writing one of the hardware accelerators in VHDL (the conversion from integer to hexadecimal ASCII – *itoh*). We also selected for acceleration the vector addition (*vadd*) and the element summation (*vsum*), from which however the accelerators were automatically translated from Java to a VHDL OPB/HWALB core. Full systems were built in the Xilinx Platform Studio using both the cores from section 4.1 and the hardware accelerators, and then synthesized, mapped, placed, and downloaded on a Virtex-II FPGA.

For evaluation we initially built two systems, one without any hardware acceleration (*plain*), where all tasks are carried out on the processor, and a system containing the three hardware accelerators just mentioned (*vadd,vsum,itoh*). We then compared these two systems from several points of view. First, the micro-code size for the accelerated system (including the code for *hwWrite*, *hwSynch*, and *hwRead*) marginally increased from 869 to 881 10-bit words (1.4%). Note that this is a one time change, regardless of the number of the accelerators.

Second, we were interested in the area increase or the device utilisation for the accelerated system. Fig. 4 presents the figures gathered from the MAP report (Xilinx ISE 6.2) for the hardware accelerator local bus (HWALB), the *itoh*, *vsum*, and *vadd* accelerators, followed by the overall device utilisation for the accelerated system, the overall device utilisation for the initial system and finally, the figure for our version of the JOP core. Note that the device utilisation grows for the accelerated system with about 75% for this particular example. For comparison, on the XC2V1000 device used for evaluation, the accelerated system still occupies a rather small percent of the available resources (28% of the total Slice Flip Flops and 14% of the total 4 input LUTs).

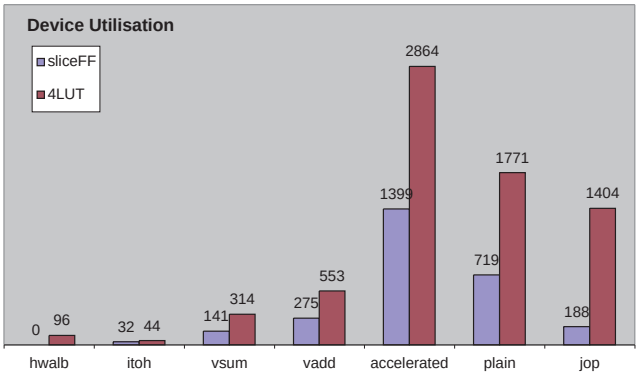

Figure 4: Device utilisation for the different hardware accelerators (*hwalb, itoh, vsum, vadd*), the full system both after (*accelerated*) and before acceleration (*plain*), and also for our version of JOP (*jop*).

Third, the program memory footprint is also affected by replacing code sequences with hardware calls. Intuitively, one expects the software to shrink as more functionality is moved to hardware. For the example we used, however, the memory footprint slightly increased from 3173 to 3198 words (0.7%). This can be explained on one hand by the properties of the sequences that were replaced, which although take a long time to execute, do not actually take many bytecodes in the executable. On the other hand, a large number of values exchanged with an accelerator (*itoh* needs 9 values and, thus, 10 hardware calls) may increase the accelerated code. Furthermore, if many of these parameters are return values, they have to be transferred back in the memory, increasing the code even more.

Finally, the fourth and most interesting figure we looked at

is the speed-up of the accelerated system. We measured this through the OPB_TIMER core, programmed to count clock cycles. In Fig. 5 we present the execution time in clock cycles for a specific application region consisting of three segments chosen for acceleration (*itoh, vadd, vsum*). Furthermore, for a more fair comparison, these are reported for a number of system architectures and design flows as follows. Besides the non-accelerated Java on JOP (*jop*) and JOP plus hardware accelerators (*jop+Hw*), we also examined a MicroBlaze-based [12] system and the same application written in pure C, compiled with *gcc*, both with no optimisation (*mb, gcc –O0*) and highly optimised (*mb, gcc –O2*). Furthermore, on the same MicroBlaze system, we also looked at a flow that uses first a Java to C translator [9] to convert the initial Java application into C, which can then further be compiled using *gcc* (*mb, java2c –O2*). We also tried to use similar memory architectures for both JOP and the MicroBlaze architectures. In particular, since JOP has a method cache, we also enabled the instruction cache (I$) on the MicroBlaze. In addition, we conducted measurements for three different memory architectures, namely block RAM (*BRAM*), double data rate SDRAM (*DDR*), and static RAM (*SRAM*). The *java2c* case for the BRAM is missing, since the generated executable could not fit in the available 64KB BRAM available on the Virtex-II. We also had problems getting *jop+Hw* run with the SRAM. All systems used a 66MHz clock, due to device speed grade and off-chip memory timing. However, the individual components (JOP, accelerators, MicroBlaze, ...) are reported by the synthesis tool as able to work even with a 100MHz clock.

To sum up, introducing the accelerators in the JOP architecture leads to an impressive performance improvement (a 3 to 6 times speed-up for the examined application). Furthermore, the application runs on *jop+Hw* slightly faster than the optimised *gcc* compiled C code on the MicroBlaze, rendering Java competitive even for embedded systems.

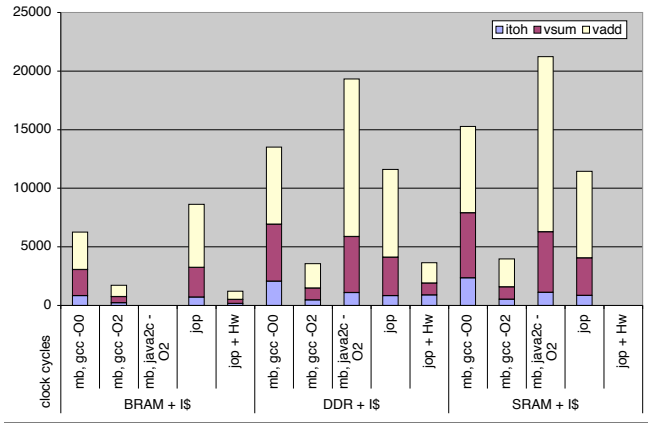

Figure 5: The execution time in clock cycles for the test application, for various architectures and flows. Solutions based on MicroBlaze and C/Java are denoted by *mb* and *gcc/java2c*. Our own solutions are denoted by *jop* and *jop+Hw*.

6. SUMMARY

The current paper described a semi-automatic approach for fast generation of application specific system, based on a Java micro-programmed core, suitable for FPGA systems-on-chip. An overview of the proposed design flow was given, followed by a discussion regarding the architectural choices, and concluded by presenting the results of employing our flow to synthesis a simple application. In particular, for that test application we observed a speed-up between 3 and 6 times, at an expense of 75% increase in the device utilisation.

We believe that the novelty and the strength of our approach resides in a combination of features, as follows. The architectural choice for integrating the hardware accelerators with the processors allows for a straight forward and rather simple design flow. Using a micro-programmed core with specific hardware call micro-programs hides the accelerator access procedure from the high-level software, while giving more opportunities for a closer interaction between the processor and the hardware functions. In addition, by keeping the accelerators out of the core execution path, the need for a specialised compiler (as necessary for ASIPs) is eliminated.

7. REFERENCES

[1] P. Andersson and K. Kuchcinski. Automatic local memory architecture generation for data reuse in custom data paths. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 21–24 2004.

[2] J. M. P. Cardoso and H. C. Neto. Fast hardware compilation of behaviors into an FPGA-based dynamic reconfigurable computing system. In *The XII Symposium on Integrated Circuits and System Design*, pages 150–153, October 1999.

[3] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computer Survey*, 34(2):171–210, 2002.

[4] M. B. Gokhale and J. M. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126. IEEE Computer Society, 1998.

[5] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedins of the International Conference on VLSI Design*, January 2003.

[6] IBM. On-chip peripheral bus, architecture specifications, version 2.1. Technical Report SA-14-2528-02, IBM, 2001.

[7] Insight Memec. http://www.insight-electronics.com/memec/iplanet/link1/virtex11mbv1000.pdf.

[8] K. Kent, H. Ma, and M. Serra. Rapid prototyping a co-designed java virtual machine. In *15th International Workshop on Rapid System Prototyping (RSP) 2004*, pages 164–171, June 2004.

[9] A. Nilsson. Compiling Java for real-time systems. Licentiate thesis, Lund Institute of Technology, 2004.

[10] M. Schoeberl. JOP: A java optimized processor. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, November 2003.

[11] SystemC. the open systemC initiative. http://www.systemc.org.

[12] Xilinx. *MicroBlaze Processor Reference Guide*, EDK v6.2 edition, June 14 2004.