

Automatic Generation and Testing of Application Specific Hardware Accelerators on a New Reconfigurable OpenSPARC Platform

Cecilia González-Álvarez^{1,2}, Mikel Fernández², Daniel Jiménez-González^{1,2},
Carlos Álvarez^{1,2}, and Xavier Martorell^{1,2}

¹ Barcelona Supercomputing Center

² Universitat Politècnica de Catalunya

{cecilia,fernande,djimenez,calvarez,xavim}@ac.upc.edu

Abstract. Specific hardware customization for scientific applications has shown a big potential to address the current holy grail in computer architecture: reducing power consumption while increasing performance. In particular, the automatic generation of domain-specific accelerators for General-Purpose Processors (GPPs) is an active field of research to the point that different leading hardware design companies (e.g. Intel, ARM) are announcing commercial platforms that integrate GPPs and FPGAs. In this paper we present a new framework with a holistic approach that addresses the challenge of design exploration of specific application accelerators. Our work focuses on a target platform consisting of a GPP with a reconfigurable functional unit. The framework includes a reconfigurable 1-core 1-thread OpenSPARC with a new programmable specific purpose unit (SPU) inside the OpenSPARC core. In order to program the SPU we have developed an automatic toolchain that profiles an application and discovers its main computing bottlenecks. With that information our toolchain is able to both design hardware specific accelerators that can be automatically mapped in the aforementioned SPU, and generate the binary code necessary to run the application using those accelerators. The OpenSPARC with the new specific application accelerators, defined in a Hardware Description Language, can then be executed and measured. Still awaiting further development, nowadays our framework is a proof-of-concept that shows that this kind of systems can be developed and programmed as easily as a GPP. In a near future it would be the source of very interesting information about the capabilities and drawbacks of those mixed GPP-FPGA systems.

1 Introduction

General purpose computing has reached a point where it is not possible to go further beyond the limits of Amdahl's law applying the same techniques all over again. The reasons for that assertion are at least two: (1) power imposes a limit to the number of CPU parts that can be running at the same time, reducing the capacity of increasing the speed of current processors; and (2) general purpose

2

processors (GPPs) are not able to cope with application-specific characteristics because of their generality.

Current trends in processor design lead to increase the number of processors on the same chip. This approach has two main variants: homogeneous and heterogeneous multicore processors. Of course, either way has its drawbacks: on one hand, homogeneous multicores may have several cores wasting computing power since their full exploitation (all cores running) depends on the runtime of the programming model used, and the programmability and expression of the parallelism of the application, if it exists. On the other hand, heterogeneous multicores reduce the problem of power consumption with power-efficient units, but the time to market of new applications is higher because of increased programming complexity. There are not known efficient mechanisms to automatically map every application or application parts to their best suited hardware. Therefore, the generation of an application-specific processor based on profiling information may help to solve the wasted power problem and to exploit the parallelism of the applications at low cost. Of course if we want to address more than one application (or set of applications) with the same hardware this can only be achieved with a reconfigurable unit. In this direction, new architectures including a GPP attached to an FPGA are being introduced by leader companies in hardware design (e.g. Intel [5], ARM [11]). This new trend can overcome several of the aforementioned problems: FPGAs can now implement very specific features needed for special applications, and then be programmed to achieve high data-level parallelism as the most suitable configuration for other applications. However these gains are achieved at the cost of putting even more stress in the programmability of the system which is not leveraged but hindered.

In this paper we present a new framework as a proof-of-concept of an automatic mechanism, transparent to the programmer, that can exploit those new architectures. The mechanism automatically: (1) identifies the application-specific hardware accelerators needed based on profiling analysis, (2) generates HDL code for those hardware accelerators and (3) modifies the binary code of the original application to run on the application-specific reconfigured architecture. In particular, the software framework presented on this paper has been build using the LLVM infrastructure. And the hardware framework uses the OpenSPARC T1 with a reconfigurable functional unit, added for the purpose of exploring and analyzing application-specific fine-grained hardware accelerators.

2 Related work

The automatic extraction of fine-grained hardware accelerators is equivalent to extracting ISA extensions. This is usually divided into two different objectives: identification of candidates, and selection of new instructions [7]. The identification of candidates involves the generation of patterns consisting of subsets of the sequence of instructions executed on the original application. That identification can be solved with an exhaustive search through the graph that represents the application. As the search can grow exponentially, some authors propose different backtracking approaches [2]. The selection of the final set of instructions may

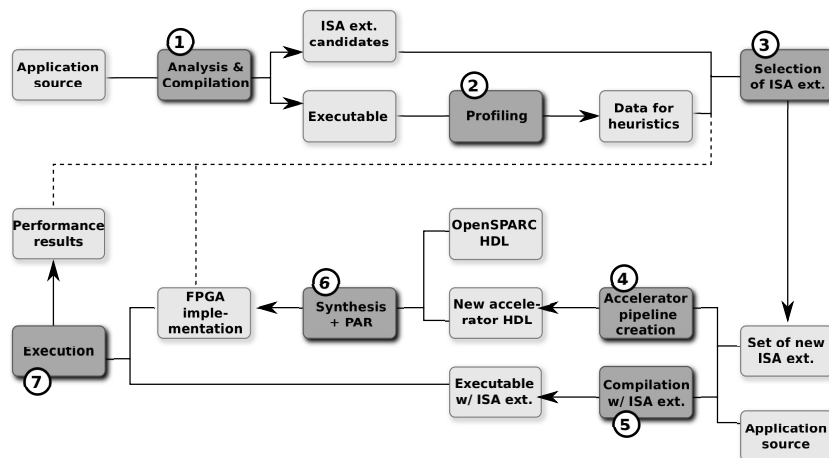


Fig. 1. Toolchain processes of the Framework.

vary depending on the metrics that are evaluated. The most popular way to address this selection is the use of heuristics, guided by a cost function [12, 4, 9, 8]. This approach might not give the optimal solution, although experimentally has been tested to be good-enough, depending on the metrics of the cost function. In order to get optimal solutions, the selection problem is often tackled with Integer Linear Programming methods [12, 1]. While previously cited works do not use real hardware prototyping, our fine-grained approach does, as well as other coarse-grained approaches, like the coprocessor in the MOLEN Architecture [10] and the runtime acceleration of a MIPS processor in [3].

3 Framework Description

The envisioned architecture of our framework consists of a single core which incorporates a new specific-purpose reconfigurable function unit (RFU) to its pipeline along with its typical functional units. That RFU is modified conveniently for each application based on a profiling analysis and some heuristics criteria. In our particular implementation, the single core is one core of the OpenSPARC T1 processor (synthetizable on a Xilinx Virtex 5 FPGA), and the RFU is implemented as an adaptation of its floating-point unit (FPU), which has been integrated into the core to better match the architecture proposed.

Figure 1 gives an overview of the toolchain presented in this paper. Dark grey boxes stand for processes, while light grey ones are the input or output data to/from processes. There are seven main processes:

1. Analysis and compilation: generates application-specific hardware accelerator candidates and an executable from the application source code.
2. Profiling: gets data from executing the application in the target architecture.
3. Selection of the hardware accelerators: generates the best set of fine-grained hardware accelerators from the set of candidates, the profiling data (before and after accelerating the application) and other metrics obtained from the bitstream generation (e.g. area, power).

4

4. Accelerator pipeline creation: creates the pipeline HDL that can execute the accelerator description from the input.
5. Compilation with ISA extensions: the new ISA is used to generate an executable that can use the new reconfigurable function unit (RFU).
6. Synthesis + PAR: generation of a bitstream from the HDL of the OpenSPARC core and the new accelerator pipeline. Its results may help with selection heuristics.
7. Execution of the program on an FPGA running the modified OpenSPARC. Results may feed the selection to help with heuristics.

Functionality and implementation of the baseline OpenSPARC platform and the toolchain of the framework are detailed in subsequent sections.

4 Generic Reconfigurable OpenSPARC Platform

OpenSPARC is an in-order pipelined RISC processor implementing the SPARC v9 ISA. The choice of this processor is due to both its availability as an open source Verilog code and the fact that can be mapped to a FPGA board.

4.1 Design Decisions

Once the OpenSPARC processor has been chosen as our base processor there are three different possible approaches in order to create a baseline platform with enough flexibility to automatically reconfigure an application-specific functional unit: (1) modifying the execution unit, (2) modifying the Cryptography unit, and (3) modifying the floating point unit (FPU).

Modifying the integer execution unit would be faster than using the FPU or the cryptographic unit for two operand instructions, but it provides our schema with no extra flexibility. The cryptographic unit might allow more flexibility if all its code was open source. As it is not, it does not seem to be easily modified to become a baseline platform that can be automatically reconfigured. Therefore, the FPU was finally chosen since it implements input and output queues as a co-processor unit. Using the same strategy than the OpenSPARC, we solve the problem of having operations with different number of cycles, allowing the integration of new accelerators in a pipeline with arbitrary latency. This is done without changes to the control unit.

4.2 In-core Reconfigurable FPU

The original FPU has three different pipelines, one for each floating point (FP) operation family implemented: add, multiply, and division. Add and multiply data paths are totally pipelined and have a fixed number of stages, while divisions take a variable time to execute (depending on its operands) and are not pipelined. Those FP operations are firstly requested by the front-end unit (FFU) to the local store unit (LSU), that is in charge of requesting operations through the crossbar to the FPU, which will finally capture those requests. The FPU only allows two source operand requests. FFU also forwards memory requests through the LSU. Simpler FP instructions, such as register to register and memory to

register movement instructions, are implemented in the FFU. Therefore, the original FPU is off-core and that makes accessing it a high latency operation.

In this work, our first modification has been to move the FPU inside the OpenSPARC core. That has been done in such a way that we minimize the changes in the rest of the OpenSPARC architecture. In particular, we have created a bridge between the FFU and the original FPU (now inside the core), and the LSU. That bridge forwards FFU memory requests to the LSU, and prepares the FFU FP requests, with the same format they were prepared by the LSU on the original OpenSPARC, to the FPU. Our second modification allows more than two source operands on the requests, introducing:

- A new register file whose registers can be used as operands in the RFU-based accelerators. This new register file is called Temporal Register File (TRF). The values stored in the TRF can be read and used by the instructions implemented using the new pipeline added to the FPU.
- Three new instructions called Temporal Move (TMV): *TMV1* to move an extra operand to a given register in the TRF. For more than one extra operand, we use *TMVR* and *TMV2*. *TMVR* moves two operands to registers 0 and 1 of the TRF initializing an index (*idx*) to 2, and *TMV2* moves two operands to registers *idx* and *idx+1* and increments ($idx \leq idx+2$).

Any new pipeline added to the OpenSPARC processor using our reconfigurable platform will work the same way as the *add* or *mul* pipelines. It would be divided in two submodules (one for the control unit and another one for the data path), and would return only one value to the SPARC core. The coding of the new pipeline has been selected from available free codes of the FPU. More details about implementation, coding and some examples can be found in [6].

5 Generation of Application Specific Accelerators

In this section we go through the process of identification, selection and generation of new accelerators for the target platform.

5.1 Identification of Application Specific Accelerator Candidates

The input of this phase is the target application source code. It can be written in any language that is supported by LLVM-GCC frontend, such as C, C++, Fortran, etc. LLVM transforms the high-level code into an intermediate representation as a direct acyclic graph (DAG) that is analyzed to identify the new accelerator candidates. This step is implemented within the target-independent Code Generator of LLVM.

LLVM Instruction Selection Analysis. The first phase of the LLVM code generator is Instruction Selection, where graphs called SelectionDAGs are created from the LLVM intermediate representation of the original source code. We decided to perform our analysis on a SelectionDAG because it is an abstract representation very close to the target architecture instructions, yet being the

process target-independent. Thus, this part of the framework can be reused in as many different testing hardware platforms as desired. We introduced our analysis step before the latest step (scheduling) of the Instruction Selection.

Analysis: Algorithm description. The analysis algorithm is executed for every basic block on the code that we want to accelerate, selected by profiling. This analysis is fed by the SelectionDAG, the list of allowed types of operations that the accelerator will execute, and the maximum number of inputs and outputs allowed for the accelerator. The analysis algorithm creates a topologically sorted matrix that contains only the allowed operation types and then fixes the maximum number of inputs and outputs. The output result of this process is the most promising accelerator candidate for the basic block being analyzed, and this feeds the next phase of the toolchain.

5.2 Selection of Best Application Specific Accelerators

Once we have got a set of accelerator candidates, the selection of the most promising accelerators is done based on performance metrics. Those metrics (e.g. frequency, area usage number of pipeline stages and/or power consumption) are used as the heuristics data of the selection algorithm. The selection behaves as follows: First, each candidate is given a score based on those metrics. Second, duplicated candidates are merged keeping their basic block information so that accelerator candidates that initially seemed unlikely to be selected may become the most promising ones because they are used in more than one basic block. Finally, the best scored accelerator is chosen to be implemented as a real hardware accelerator.

One of the interesting points of the selection is that the heuristics can be configured to accept any type of metric that may be considered relevant for the accelerator selection. The most straightforward metric is the timing or cycle count for each basic block. This indicates the frequency of execution for a given part of the code. Profiling counters are used to get this kind of information, obtained through a previous execution of the application in the SPARC processor. In addition, we can feed the heuristics with data from later stages of previous runs of the toolchain; for instance, area occupancy of the new accelerator, or the number of stages of the new pipeline.

5.3 Generation of Application Specific OpenSPARC Accelerators

In this section we present a case study to describe the results of our toolchain: (1) the new assembler code of the application using the generated accelerators, and (2) the hardware description of the functionality of the RFU. We show how the FIR example in Figure 2(a) is accelerated, whose original assembler code is shown in Figure 2(b). Note that we have chosen an integer application to simplify the generation of the hardware accelerator. However, that should be integrated into the RFU with the corresponding execution overhead, explained in Section 7.

First of all, our toolchain will identify and select the sequence of instructions inside the loop body of Figure 2(a) to be executed in an accelerator. This

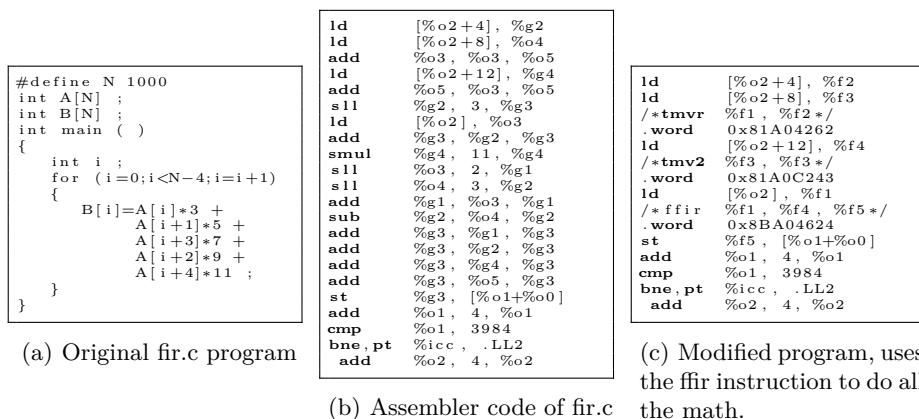


Fig. 2. Source(a), compiler generated assembly(b), transformed code(c), for the FIR algorithm.

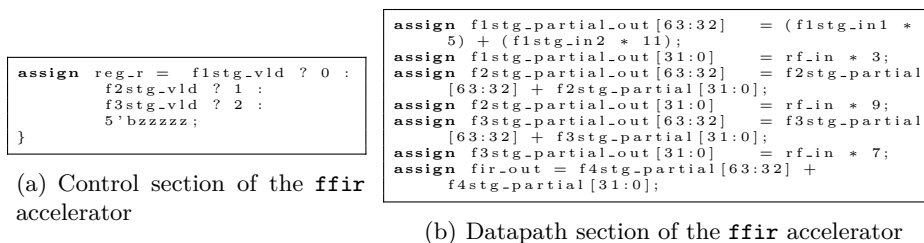


Fig. 3. Verilog code for the ffir hardware accelerator.

sequence is an ideal accelerator candidate as it includes several multiply operations that could be done all in parallel, depending on the hardware resources and width and number of memory ports. Once the selection is done, the toolchain generates new assembler code as shown in Figure 2(c) using the accelerator (i.e. `ffir`) instead of the original sequence of instructions. As commented in Section 4.2, this new code also includes the register movement instructions (i.e. `tmv*`) in order to be able to operate with more than two registers.

On the other hand, the verilog code of the hardware for the new accelerator should be generated and included into the reconfigurable OpenSPARC platform, in particular into the RFU. To perform this work two types of codes are used: (1) control code that indicates which registers should be read in each stage as shown in Figure 3(a), and (2) datapath code that indicates which are the operations to be done in each stage. As it can be seen in Figure 3(b), in the case of the `ffir` accelerator there are four stages, and just `mul` operations (that can be optimized). Therefore, with the baseline platform created, the hardware generation difficulty only falls into the implementation of the operations to be done rather than in the integration of the new accelerator.

6 Experimental Setup

The modified OpenSPARC platform has been synthesized on to a Xilinx Virtex 5 110T, using the Xilinx Synthesis Technology (xst) 11.1, and the OpenSPARC T1 1.7 release. `sims` simulator, included in the OpenSPARC T1 1.7 package, has been used to analyze the performance of the original and modified OpenSPARC.

The fine-grained hardware accelerator identification has been implemented within LLVM 2.8, meanwhile the hardware accelerator candidates has been stored in a Mysql database accessed with the Django 1.1.1 Python framework.

Three different integer application examples have been evaluated in this paper: FIR, EDGE detection, and Stencil 3D [6].

7 Results

In this section we analyse the modified and the original OpenSPARC in terms of area, cycle time, number of instructions executed, cycles and overhead.

Element	Baseline	50stg	FIR	EDGE	Stencil3D
# of Slice Registers	23212/69120	35368	23725	24186	25164
# of Slice LUTs	36971/69120	57345	37677	37996	39297
# used as Logic	34710/69120	55089	35427	35629	36984
# used as Memory	2261/17920	2256	2250	2367	2313
# used as RAM		1786	1686	1778	1778
# used as SRL		475	570	472	589

Table 1. Slice logic utilization for the sparc module.

First, Table 1 shows the slice logic utilization for the SPARC module with 5 different configurations: (1) the FPU without accelerators is integrated into the SPARC module (baseline); (2) an empty accelerator with 50 stages is integrated (50stg) in the FPU; (3) the FIR accelerators are in the FPU; (4) accelerators for EDGE are in the FPU; and (5) Stencil3D accelerators are in the FPU. The amount of registers used in the FPU with FIR and EDGE accelerators is half-way between the baseline and the 50 stage pipeline, as shown in the slice usage increment of the SPARC module. That reflects that the number of slice registers used is proportional to the number of stages. Indeed, the number of slice LUTs used is larger when integrating FIR and EDGE accelerators than for the 50-stage pipeline because they implement new functional units and the 50-stage pipeline is just an empty pipeline. Integrating the Stencil3D accelerator increases the used slice LUTs and slice registers compared to FIR and EDGE extensions. That is because the FPU with Stencil3D needs a 16-entry TRF, compared to the 8-entry TRF used by the FPU with FIR, EDGE, and the 50 stage pipeline. As it can be seen there is enough space for design exploration on new accelerators.

Regarding to the cycle time, the SPARC module, with the FPU integrated, has a maximum frequency of $93MHz$ which is slightly faster than the original SPARC module with the FPU separated ($91MHz$). In any case, the maximum frequency obtained would depend on the amount of work to be done in each stage of the new accelerators.

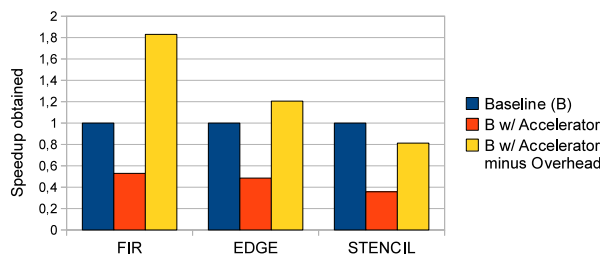


Fig. 4. Speedup for the analyzed codes assuming extra and no extra latency.

Figure 4 shows the speedup for the accelerated code compared to the original code running in the baseline architecture. For each application there are three columns. First column corresponds to the baseline architecture running the application (normalized to 1). Second column is obtained by timing the new code using the accelerators. Third column subtracts the execution overhead due to current limitations of our platform, to the second column. Those limitations are two: (1) there are still some extra cycles that have to be paid to reach FPU although we have integrated the FPU within the SPARC core, and (2) the accesses to the floating point registers in the SPARC core are more expensive than accesses to integer registers. That could be solved allowing TMV instructions to access the integer register file of the core, however this improvement is still not implemented in our actual version. Therefore, considering the third column, the best speedup achieved is for the `ffir` accelerator. Using this accelerator, the number of instructions in the body loop has been reduced from 22 to 12 instructions, and in addition, the `ffir` accelerator only has 4 cycle latency. In the case of the `EDGE` and `STENCIL` applications, the number of instructions are not reduced as much as `FIR`, and the latency of the accelerators is bigger, 8 and 10 respectively. As it can be seen the final speedup is completely application specific and another iteration of profiling may be necessary to fine-tune these issues.

8 Conclusions and Future Work

In this work we have presented a new framework that automates the inclusion of application specific hardware accelerators in a standard GPP as the OpenSPARC. Our toolchain is able of automatically profile new applications, identify their computing bottlenecks, generate new hardware accelerators that are integrated to the reconfigurable unit of our platform, and generate assembler code that uses those accelerators. The modified OpenSPARC has been mapped to a FPGA, and tested using the HDL testing infrastructure of the OpenSPARC.

Results show that our proposed system can achieve a speed-up of up to 1.83 with an increase in size over the original processor of up to 7%. Indeed, these figures are expected to be improved as the automatic system evolves, since there is still a lot of work to be done to overcome the limitations of our system. For instance, providing more than one result, introducing vector processing and data reuse support, using larger FPGAs to synthesize, place and route the OpenSPARC system at higher clock frequencies, etc.

Finally, to the best of our knowledge, this is the first automatic and flexible framework that can be used to evaluate the real behaviour of new hardware accelerators included in standard processors. Furthermore, the whole toolchain can be applied to new commercial architectures that join a GPP to a relatively large FPGA to provide fast deployment of new applications.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (TIN2007-60625, CSD2007-00050, TIN2006-27664-E), the Generalitat de Catalunya (2009-SGR-980), the European Commission in the context of the EnCORE project (FP7-248647) and the HiPEAC2 Network of Excellence (FP7/IST-217068). The FPGA was provided by the Xilinx University Program.

References

1. Kubilay Atas, Can Ozturan, Gunhan Dunder, Oskar Mencer, and Wayne Luk. CHIPS: Custom Hardware Instruction Processor Synthesis. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 27(3):528–541, 2008.
2. Kubilay Atas, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. *Int. J. Parallel Program.*, 31(6):411–428, 2003.
3. Antonio Carlos S. Beck, Mateus B. Rutzig, Georgi Gaydadjiev, and Luigi Carro. Transparent reconfigurable acceleration for heterogeneous embedded applications. *Conf. on Design, Automation and Test in Europe (DATE)*, pages 1208–1213, 2008.
4. Nathan T. Clark and Hongtao Zhong. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.*, pages 1258–1270.
5. Intel Corp. Intel AtomTM Processor E6x5C Series-Based Platform for Embedded Computing. <http://edc.intel.com/Link.aspx?id=3961>.
6. Mikel Fernandez. Master thesis: Hardware platform to test new isa extensions. <http://www.mikelfernandez.com/MT-MikelFernandez.pdf>, September 2010.
7. Carlo Galuzzi and Koen Bertels. A framework for the automatic generation of instruction-set extensions for reconfigurable architectures. In *ARC '08: 4th international workshop on Reconfigurable Computing*, pages 280–286.
8. Diviya Jain, Anshul Kumar, Laura Pozzi, and Paolo Ienne. Automatically Customising VLIW Architectures with Coarse Grained Application-specific Functional Units. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems*, 2004.
9. Bhuvan Middha, Varun Raj, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne. A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 2–7, 2002.
10. Stamatias Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, Student Member, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, pages 1363–1375.
11. Xilinx. Xilinx extensible processing platform. <http://www.xilinx.com/technology/roadmap/processing-platform.htm>.
12. Pan Yu and Tulika Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 723–728, New York, NY, USA, 2004. ACM.