Fast Instruction Set Customization

Edson Borin* Felipe Klein* Nahri Moreano*† Rodolfo Azevedo* Guido Araujo*

* IC-UNICAMP – Brazil † DCT-UFMS – Brazil

{borin,felipe.klein,nahri,rodolfo,guido}@ic.unicamp.br

Abstract

This paper proposes an approach to tune embedded processor datapaths toward a specific application, so as to maximize the application performance. We customize the computation capabilities of a base processor, by extending its instruction set to include custom operations which are implemented as new specialized functional units. We describe an automatic methodology to select the custom instructions from the given application code, in a way that there is no need of compensation code or other modifications in the application, simplifying the code generation. By using the ArchC architecture description language, fast compilation and simulation of the resulting customized processor code are achieved, considerably reducing the turnaround time required to evaluate the best set of custom operations. Experimental results show that our framework provides large performance improvements (up to 3.6 times), when compared to the base general-purpose processor, while significantly speeding up the design process.

1. Introduction

In recent years, the market of embedded processors has grown drastically. General-purpose processors frequently do not offer the necessary performance, at the low cost required by embedded systems, because the instruction set architecture (ISA) of these processors is designed to provide a reasonable level of performance for a wide variety of applications. Since an embedded processor usually runs in a limited application domain, a better cost/performance ratio can be achieved by customizing the system toward the application(s) it needs to execute.

The synthesis of application-specific instruction set processors (ASIPs) traditionally involved the generation of a complete ISA for the targeted application. However, this full-custom solution is too expensive and has long design turnaround times. We would rather use an available processor and extend its instruction set with custom operations for a specific application or domain. These operations are implemented in hardware in the form of specialized functional units. The goal of such processor extensions is to optimize the application performance, while keeping the cost of the processor down and meeting time-to-market constraints.

The problem of determining application-specific instruction set extensions consists in detecting, in the application, clusters of primitive operations which, when implemented in hardware as a single custom instruction, maximize its performance. Given an application, the number of possible specialized instructions grows exponentially with the program size, leading to a large design space, while the performance gain achieved with them varies significantly. Thus, the process of customization should be as automatic as possible.

In this work, we propose a framework to automatically find specialized instruction set extensions for a base processor, given the intended application. Our approach identifies clusters of operations that can be replaced by custom instructions, in a way that there is no violation of data or control dependencies and no need of compensation code or other modifications in the application, simplifying substantially the code generation. We also provide automated support for compilation and simulation of the resulting customized processor code, in a way that the specialized instructions are automatically used by our software tool chain. This allows the designer to efficiently explore the design space, by searching for the best performance for the target application, given the architectural constraints.

Experiments with several benchmark programs indicate that custom processors generated using our framework can result in large improvements in application performance (up to 3.6 times) compared to a base general-purpose processor, while speeding up the design process significantly.

This paper is divided as follows. Section 2 describes previous work related to instruction set customization. Section 3 presents our design flow and details the framework we propose to solve this problem. In Section 4 a set of experiments is described to support the efficiency of proposed approach. Finally, Section 5 concludes the work.

2. Related Work

Early works on architectural synthesis for ASIPs study the automatic generation of a complete custom instruction set for a given application domain [19, 11]. Recently, the focus has moved to the synthesis of application specific instruction set extensions. The selection technique presented in [5] maximizes the reuse of specialized instructions and minimizes the number of selected instructions. It does not directly maximize performance and can lead to the generation of small clusters. In [1] the authors use pattern matching and generation to identify frequently occurring patterns in an execution trace. Since this trace represents the dynamic behavior of the application it can be really large, thus reducing the efficiency of the technique. The approach in [2] imposes architectural constraints during cluster construction, pruning invalid clusters, but it is not able to completely avoid the exponential worst-case execution time.

The templates are selected in [18] using performance, energy or both as criteria, but in the experiments they included only a limited class of control constructs, i.e., not all "if" and "switch" statements were handled. In [6] a dataflow graph exploration approach is used to identify subgraphs subject to constraints but the custom instructions are limited to basic block boundaries. The framework in [4] selects a preconfigured extensible processor and code segments for extensible instructions, which are generated exhaustively enumerating all sequences of consecutive statements in critical functions. The selected segments are manually searched in an instruction library and, if not found, implemented also manually. In [8] ISA extensions are generated by the Ten-

0-7803-8631-0/04/$20.00 ©2004 IEEE.

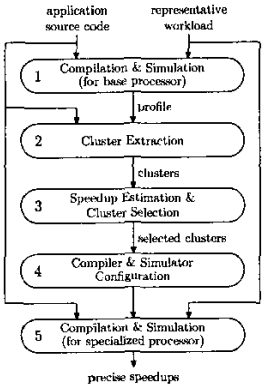


Figure 1: Steps of design flow

silica Instruction Extension System, including instructions and register files, and using VLIW and vectorization techniques. The custom instructions corresponding to several simple operations were limited to basic block boundaries.

Research in reconfigurable computing investigates the identification of application sections that are mapped to a reconfigurable fabric. In [12] the authors combine template matching and generation based on the recurrence of patterns, i.e., the frequency of operation type successions, rather than on frequency of execution. This led to small templates (such as pairs of operations) which were limited to basic block boundaries. The template generation approach in [3] exploits parallel templates (in which operations can be executed in parallel), besides sequential ones, but they do not extend across basic block boundaries. In both works, template matching is based on graph isomorphism. A software tool is presented in [13] to support the implementation of user-definable instructions in a FPGA, and it relies on manual identification of the instructions from the application.

3. Design Flow

In this section we present our methodology to identify specialized ISA extensions. The main steps of our design flow are shown in Figure 1 and detailed in the remaining of the section. The key design principles behind our framework are automation and efficiency, as we will describe.

3.1. Platform

In our approach, the applications can be written in C or C++ high-level language. We have adapted the well-known GCC tool suite [17] such that it extracts, from the application, clusters of operations which are candidates for hardware implementation. The compiler can also generate the application executable code for both the base processor and the specialized one. In the last case, it automatically uses the custom instructions, given the selected clusters.

The user does not need to modify the application source code neither to program in assembly language, in order to use the specialized instructions. The application can be compiled with the usual compiler options, including the optimizing "-O3" option. Moreover, there is no need to recompile the GCC tool every time a new instruction is added.

We have used a SPARC V8 processor as our base processor, which is extended with the specialized functional units and custom instructions. These functional units are included into the processor datapath and their input operands and results are, respectively, read from and written to the SPARC integer register file. The custom instructions can consume a single cycle or be a multi-cycled instruction. Both base and specialized processors are simulated using cycle-accurate models developed with the ArchC architecture description language [16]. The base processor model is adapted automatically to include the custom instructions.

3.2. Cluster Extraction

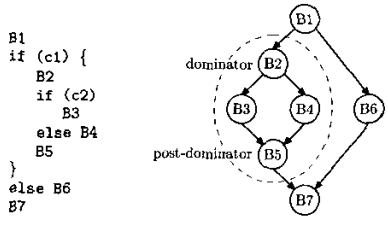
Starting with the application source C/C++ code and a representative workload, we compile and simulate the application using the base processor model. We then measure the performance of the application for the base processor and we compute additional profile information. Using this information, we can identify the performance-critical sections of the application, i.e., inner-loops and functions that consume most of the program execution time.

We feed the application source code into the modified GCC compiler again in order to automatically extract all valid clusters from the sections identified previously. A cluster corresponds to a piece of the application code and is composed by primitive operations such as *add* and *load*. Therefore, our approach does not require the programmer to manually select which parts of the application should be implemented as custom instructions. Generating clusters only for the performance-critical sections of the application reduces significantly the number of extracted clusters and is very important for the efficiency of the framework.

The clusters are extracted from GCC intermediate representation (IR). Using IR instead of machine instructions permits us to extract the cluster after all machine-independent code optimizations, but before register allocation and machine-dependent optimizations. Whenever possible, automatic function inlining [15] is applied. The application code corresponding to a cluster is not limited to basic block boundaries and can contain control constructs, such as "if-then", "if-then-else", "switch", and "goto".

Valid clusters are those that can be replaced in the application code by a specialized instruction, without violating data or control dependencies. That is, the compiler is able to find an instruction schedule which respects the application semantics. Our cluster extraction algorithm uses dominance and post-dominance relations between the operations in order to guarantee that only valid clusters are extracted. An operation d dominates an operation i (d *dom* i) if every possible execution path from the beginning of the procedure to i includes d. An operation p post-dominates an operation i (p *pdom* i) if every possible execution path from i to the end of the procedure includes p [15].

We illustrate these concepts with an example. Given the section of code shown in Figure 2, the corresponding control-flow graph (CFG) represents all possible execution paths in this code. We can see that the operations in basic block B2 dominate all operations in B3, B4, and B5. Also, the operations in B5 post-dominate all operations in B2, B3, and B4. We ensure that a valid cluster is produced by requiring that the initial (final) basic block of the cluster dominates (post-



(a) Code (b) CFG

Figure 2: Dominance criteria for cluster generation

dominates) all blocks in CFG from the initial one until the final block is reached, and reversely, from the final one until the initial block is reached. For example, blocks B2, B3, B4, and B5 constitute a valid cluster. Cluster granularity can also vary. In our example, another valid cluster is formed by all basic blocks in the figure, given that B1 (B7) dominates (post-dominates) all remaining blocks. Each block taken isolated also constitutes a valid cluster. Moreover, we can also generate clusters that do not necessarily include all operations from the dominator and post-dominator blocks.

Figure 3 presents our cluster extraction algorithm, which is applied to each performance-critical section identified in the application. The dominance tests on basic block B_k (marked with (*) in the algorithm) ensure that the execution of the cluster always starts and ends on B_i and B_j, respectively. This guarantees that only valid clusters are generated. In the special case the section has no loops or it is a structured inner-loop, these tests are not necessary. This algorithm has a polynomial time complexity with respect to the number of basic blocks of the input section.

The main advantage of this extraction technique is that the cluster can be entirely replaced by the custom instruction, without violating data or control dependencies. Compared to strategies involving superblocks or hyperblocks, our approach may lose some clusters but it simplifies substantially the code generation, because it eliminates the need of compensation code or modifications in the application.

Each cluster is automatically transformed into a control/data-flow graph (CDFG), where the vertices represent the primitive operations. The data dependencies between the operations are obtained from the use-definition chains [15] kept by GCC. We handle control dependencies by applying if-conversion, inserting selectors into the CDFG, and using predication only for memory operations. This is done using the control-dependence graph (CDG) [15], so we can handle both structured and non-structured (with "goto") application code. Besides, our methodology is not limited to tree-shaped clusters, what would preclude many potentially interesting clusters, e.g., clusters in which parallel operations share operands, thus reducing the number of input data to the custom instruction.

3.3. Cluster Selection

After identifying a number of clusters in the application, we want to know how useful their hardware implementation would be. For each extracted cluster, we estimate the speedup achieved if it is replaced by a specialized instruction. In this step of the design flow, we only estimate the

```
Algorithm extract_clusters
  Input: section S with basic blocks B
  Output: set L of clusters
  Build CFG, dominance and post-dominance trees of S
  L ← ∅
  for each B_i, B_j ∈ S | B_i dom B_j and B_j pdom B_i do
    Cluster ← ∅ ; valid ← true
    /* Depth first search (DFS) from B_i to B_j in CFG */
    for each B_k reached in DFS and while valid do
(*)   if B_i dom B_k and B_j pdom B_k then
        Cluster ← Cluster ∪ {B_k}
      else valid ← false
    /* Reverse DFS (R-DFS) from B_j to B_i in CFG */
    for each B_k reached in R-DFS and while valid do
(*)   if not (B_i dom B_k and B_j pdom B_k) then
        valid ← false
    if valid then L ← L ∪ {Cluster}
```

Figure 3: Cluster extraction algorithm

speedups rather than generate and simulate the specialized processor. The goal is to make the framework more efficient, given that we have not selected a small set of clusters yet.

In order to estimate the application speedup with respect to a cluster, we need to measure the number of cycles consumed by the software and hardware implementations of that cluster. From the profile information, we get the number of cycles consumed by the cluster instructions when the application was executed in the base processor. In order to determine the number of cycles consumed by the specialized functional unit which implements the cluster, we schedule its CDFG. We use a list-scheduling algorithm, which exploits chaining of operations and uses delays of primitive operations obtained from a component library. Resource constraints are imposed during scheduling, limiting the number of hardware blocks available to the specialized functional unit, as well as, the number of registers (of the base processor datapath) that can be read and written at each cycle.

The application speedup with respect to a cluster C is estimated using Equation 1, where $cycles_{base}$ is the total number of cycles consumed by the application when executed in the base processor. The number of cycles saved using C corresponds to difference between the total number of cycles consumed by the software and hardware implementations of C (considering all executions of the cluster), as shown in Equation 2. The total number of cycles of the software version of C is obtained from the profiling information, while the total number of cycles of the hardware implementation is determined using Equation 3, where $freq(C)$ corresponds to the frequency of execution of the cluster (also obtained from the profiling information). Finally, $delay_{HW}(C)$ corresponds to the delay (in cycles) of the hardware implementation of C, established by our scheduling step.

$$speedup = \frac{cycles_{base}}{cycles_{base} - cycles_{saved}(C)} \quad (1)$$

$$cycles_{saved} = cycles_{SW}(C) - cycles_{HW}(C) \quad (2)$$

$$cycles_{HW}(C) = freq(C) \times delay_{HW}(C) \quad (3)$$

Since speedup estimation is very fast, we evaluate several clusters with different resource constraints, exploring a large portion of the design space. Then we select the clusters that provide the best speedups among all extracted clusters.

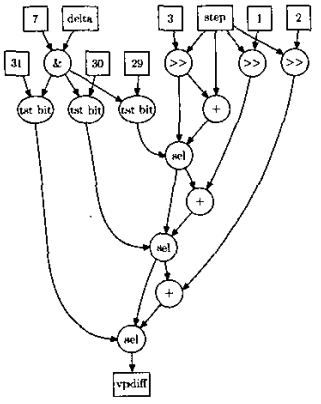
The generated clusters are not limited to sequences of few operations and our cluster selection approach is based on the

```
...
delta = delta & 7;
vpdiff = step >> 3;
if ( delta & 4 ) vpdiff += step;
if ( delta & 2 ) vpdiff += step>>1;
if ( delta & 1 ) vpdiff += step>>2;
...
```

(a) Source code

```
instr_behavior(specInstr) {
   ...
   s1 = readReg(rs1);
   s2 = readReg(rs2);
   ...
   t1 = s1 & 7;
   t2 = s2 >> 3;
   t3 = s2 >> 1;
   ...
   writeReg(d1, t15);
   setCycles(3);
   update_pc();
}
```

(c) Instruction simulator code



(b) CDFG

Figure 4: Cluster from ADPCM decoder and instruction behavior in the simulator

speedup they provide, and not on their frequency of occurrence. Therefore, our framework does not include a pattern matching step, in which different occurrences of each cluster are searched in the application. This step would require a subgraph isomorphism test, which is a NP-complete problem [7] and may consume a long execution time.

3.4. Specialized Processor Generation

For each selected cluster, we compile the application generating the executable code for the specialized processor. We automatically modify the application IR, so the compiler is able to replace the cluster operations by the new custom instruction. It is not necessary to modify the application source code nor to re-generate the GCC. Also, a specialized processor simulator with the custom instructions is automatically generated from the base processor model.

Given the CDFG corresponding to a selected cluster, our framework automatically generates the custom instruction behavior using ArchC. The number of cycles consumed by the new instruction is also set. The generated code is included into the base processor model, producing the specialized processor simulator. Figure 4 shows the source code and CDFG corresponding to a cluster extracted from the ADPCM decoder application of MediaBench benchmark [14]. The figure also shows partially the code generated to describe the corresponding custom instruction (specInstr).

Finally, the application is executed using the new simulator and we measure the performance of this application for the specialized processor. Then we compute the exact speedup achieved when the selected clusters are implemented as custom instructions. Notice that only the selected clusters go through the simulation step, thus reducing the size of design space and the design turnaround time.

The achieved speedup is mainly due to the hardware implementation of the cluster, which is more efficient than the software one because it exploits instruction-level parallelism and chaining. Moreover, it reduces the register pressure, and consequently, the memory spills, because temporary values inside the clusters are not allocated to the base processor registers any more. Also, given that a set of original instructions is replaced by one custom instruction, the number of instruction cache misses can be reduced.

Notice that our framework allows the use of different hardware implementations of the cluster. For example, a loop cluster can be implemented in a pipelined way. During the specialized processor generation, it is only necessary to set the number of cycles consumed by the custom instruction.

4. Experimental Results

The framework presented above was applied to programs from the MediaBench [14] and MiBench [9] suites. All applications were compiled with the maximum level of optimization (option "-O3"). We evaluated the speedup achieved by each cluster extracted from each application, under different resource constraints. These constraints limit the number of hardware blocks available for the specialized functional unit, which are arithmetic units (AU), logic-shift units (LSU), multipliers (Mul), multiplexors, register file read and write ports, and memory read/write ports.

4.1. Specialized Processor Performance

The estimated speedups achieved by the specialized processors are shown in Figure 5. Each curve corresponds to one cluster selected from an application and shows the speedups obtained with the inclusion of the corresponding custom instruction, under several constraints. If two clusters are totally disjoint, they can both be implemented as custom instructions, further improving performance. Each cluster is labeled with the number of its dominator and post-dominator basic blocks. However, the clusters may have been extracted from different program functions.

In the experiments we set the resource constraints so as to fix the number of multiplexors and memory ports to four and one, respectively, and vary the number of AUs, LSUs, and Muls. For example, the constraint 2/3/1 represents two AUs, three LSUs, and one Mul. We also set to two and one, respectively, the maximum number of registers (of the base processor datapath) that can be read and written at each cycle. Since we insert the specialized functional unit into the datapath of the base processor (which in our experiments

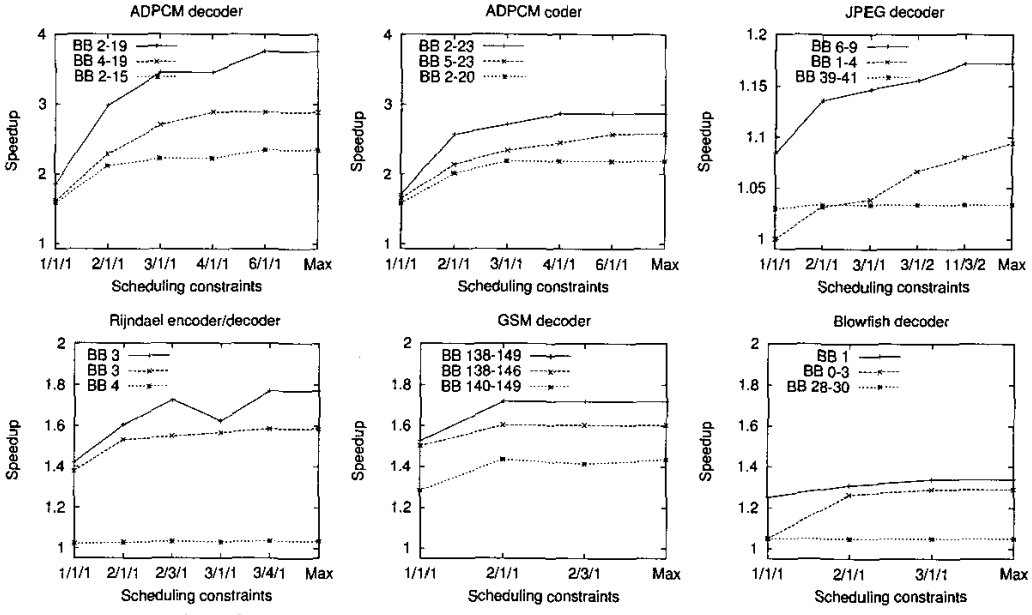


Figure 5: Speedup of specialized processor for each cluster under resource constraints (AU/LSU/Mul)

is a SPARC V8), the memory and register constraints must conform to the datapath restrictions. It is important to note that this is not a limitation of the framework and different constraints can be used with other base processors. We also evaluated the clusters under loose constraints (represented as Max in the figure), in which there were unlimited AUs, LSUs, and Muls, but the remaining resources were fixed with the same restrictions described above.

The figure shows speedups ranging from 1.1 to 3.7. Notice that, for some applications, large speedups were achieved with only one custom instruction, even with very strict constraints. Also, for almost all clusters, even with loose constraints, the speedup can not be further improved. But, if the constraints for read and write register and memory ports were relaxed, even better speedups would be possible.

Both ADPCM encoder and decoder have only one inner-loop with several basic blocks and few memory access. Since these loops consume most of the program execution time, reducing the cycles spent in the loops reflects directly to the total execution time. The best speedup is achieved under the constraints 6/1/1 because several comparison operations were produced by the if-conversion technique. The best cluster for each program corresponds to the whole loop body.

The Rijndael encoder and decoder have symmetric inner-loops with high execution frequency and several shift and logical operations, but also many memory reads. Here we have the interesting situation in which the best clusters for both applications are identical and achieve the same speedups, so the same specialized processor can be used to encoder and decoder programs. The maximum speedup is nearly 1.8 and it is not further improved due to the memory access, which brings no gain when implemented in hardware. The same situation on the memory reads also happens in the cluster BB 1 from the Blowfish decoder.

The cluster BB 138-149 corresponds to the body of the most executed loop from the GSM decoder. Due to the large number of shift and logical operations in the cluster, it is efficiently implemented in hardware, and brings the speedup of 3.5 with respect to the loop execution time. However, according to Amdahl's law [10], since the application has other performance-critical sections, a speedup of 1.7 is obtained when considering the complete program. The speedup of cluster BB 140-149 decreased under constraints 2/3/1, even though the number of LSUs increased. Since the heuristic list scheduling uses a priority function to resolve resource contention, it delayed a critical path operation.

The best cluster from the JPEG decoder has many memory, arithmetic, and multiplication operations, which have long delay and prevent the scheduler from exploiting chaining. Given that the three clusters shown for this application are disjoint, we can implement all of them as custom instructions and obtain a speedup of 1.36.

4.2. Framework Evaluation

We also evaluated the efficiency of the proposed framework with respect its execution time. Table 1 shows the execution time of each step of our design flow (enumerated in Figure 1) for a set of benchmarks, using a standard Pentium IV/Linux desktop. Even though these execution times can change significantly by using a different platform, they give an overall idea of the time consumed at each step. As shown, with the exception of steps 1 and 5 which require cycle-accurate simulation, all steps took only a few seconds.

In step 2 all valid clusters from all performance-critical sections were extracted. The speedup provided by each

Table 1: Execution time of design flow steps

Application	Step 1	Step 2	Step 3	Step 4	Step 5
ADPCM dec.	22s	0s	0s	5s	3s
ADPCM enc.	27s	0s	0s	4s	4s
Rijndael dec.	1m25s	1s	0s	5s	32s
Rijndael enc.	1m26s	0s	0s	5s	35s
JPEG dec.	15s	21s	27s	44s	13s
GSM dec.	4m10s	4s	11s	11s	1m35s
Blowfish dec.	1m43s	1s	0s	5s	50s

Table 2: Estimated and simulated speedups

Application	Estimated speedup	Simulated speedup
ADPCM dec.	3.75	3.60
ADPCM enc.	2.86	2.63
Rijndael dec.	1.77	1.88
Rijndael enc.	1.77	1.91
JPEG dec.	1.17	1.23
GSM dec.	1.72	1.66
Blowfish dec.	1.34	1.34

valid cluster was estimated for a set of different resource constraints in step 3. The quick estimation enabled us to evaluate many clusters with different resource constraints, searching for the best cluster with the minimum resources. Given that the simulation steps were longer, only the chosen best solution was simulated in step 5. Both steps 1 and 5 involved the simulation of the application, but step 1 was longer because it also generated profiling information.

Finally, we need to ensure that the estimated speedups approach the real speedups achieved. Table 2 compares the estimated and simulated speedups obtained for the best cluster of each program. The estimated speedups were very close to the simulated ones for most applications. Given that clusters are extracted after all machine-independent optimizations have been performed and before register allocation and machine-dependent optimizations, the estimated performance of the software implementation of the cluster can be slightly different from the real performance.

5. Conclusions

This paper presented a methodology to automatically identify specialized instruction set extensions for a base processor, given the intended application, with the goal of generating an ASIP which optimize its performance. The clusters corresponding to the custom instructions are extracted from the application using a dominator/post-dominator criterion which eliminates the need of compensation code and ensures that the compiler is able to schedule the new instructions. An efficient speedup estimation enable the selection of the best clusters and their evaluation under different resource constraints. The implementation of the new instruction is automatically generated using an architecture description language and included into a base processor model, producing a specialized processor simulator.

Experiments were performed to evaluate the efficiency of the framework and specialized processors were generated for a set of benchmarks. For several applications, large speedups (ranging from 1.2 to 3.6) were achieved with only one custom instruction and very strict resource constraints. The software tools which implement the design process took less than 1-2 minutes for the steps requiring simulation, and only a few seconds for the remaining steps. The speedup estimation technique produced very accurate results when compared to the speedups obtained through simulation, allowing an efficient exploration of the design space.

6. Acknowledgments

This work was partially supported by fellowship grants (CNPq 301731/2003-9 and 3821203-0, CAPES, and FAPESP 02/08139-3), and Mindspeed Technologies and FAPESP (00/15083-9) research awards.

7. References

[1] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES*, pp. 61–66, 2001.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, pp. 256–261, 2003.

[3] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, pp. 262–269, 2002.

[4] N. Cheung, S. Parameswaran, and J. Henkel. INSIDE: Instruction selection/identification & design exploration for extensible processors. In *ICCAD*, pp. 291–297, 2003.

[5] H. Choi, J. Kim, C. Yoon, I. Park, S. Hwang, and C. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–614, 1999.

[6] N. Clark, H. Zhong, W. Tang, and S. Mahlke. Automatic design of application specific instruction set extensions through dataflow graph exploration. *International Journal of Parallel Programming*, 31(6):429–449, 2003.

[7] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman and CO., 1979.

[8] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES*, pp. 137–147, 2003.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, pp. 3–14, 2001.

[10] J. Hennessy, D. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2002.

[11] I. Huang and A. Despain. Synthesis of application specific instruction sets. *IEEE TCAD*, 14(6):663–675, 1995.

[12] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM TODAES*, 7(4):605–627, 2002.

[13] A. La Rosa, L. Lavagno, and C. Passerone. Hardware/software design space exploration for a reconfigurable processor. In *DATE*, pp. 570–575, 2003.

[14] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *MICRO*, pp. 330–335, 1997.

[15] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[16] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-based architecture description language. Accepted for publication at the *16th Symposium on Computer Architecture and High Performance Computing (SBAC)*, 2004. See http://www.archc.org.

[17] R. Stallman. *GNU Compiler Collection Internals*, 2002.

[18] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE TCAD*, 23(2):216–228, 2004.

[19] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on High-level Synthesis*, pp. 11–16, 1994.