Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints

Kubilay Atasu
Kubilay.Atasu@epfl.ch

Laura Pozzi
Laura.Pozzi@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Processor Architecture Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
Lausanne, Switzerland

ABSTRACT

Many commercial processors now offer the possibility of extending their instruction set for a specific application—that is, to introduce customised functional units. There is a need to develop algorithms that decide automatically, from high-level application code, which operations are to be carried out in the customised extensions. A few algorithms exist but are severely limited in the type of operation clusters they can choose and hence reduce significantly the effectiveness of specialisation. In this paper we introduce a more general algorithm which selects maximal-speedup convex subgraphs of the application dataflow graph under fundamental microarchitectural constraints, and which improves significantly on the state of the art.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Algorithms, Performance, Design

Keywords

Customisable processors, ASIPs, Instruction-set extensions, Codesign

1. INTRODUCTION

In the last decade, research in design methodologies for system-on-chip processors has been mainly revolving around the synthesis of *Application Specific Instruction-set Processors (ASIPs)*. This involved the automatic generation of complete instruction sets for specific applications ([9], [16], [10]). In that context, the goal is typically to design an instruction set which minimises some important metric (e.g., run time, program memory size, execution unit count).

More recently, the attention has shifted toward extending generic processors with units specialised for a given domain, rather than designing completely custom processors. The goal of such processor extensions is typically to optimise performance in an application domain without incurring the area and energy cost of top-notch superscalar or multithreaded processors. Many readily extensible processors exist today both in academia (e.g., [4]) and industry (e.g., [8], [17], [7], [6]). The important motivation toward specialisation of existing processors versus the design of complete ASIPs is to avoid the complexity of a complete processor and toolset development. Instead, an available and proven processor design and its extensible toolset can be leveraged: design efforts must focus exclusively on the special datapath.

We believe that *it is fundamental to generate the required instruction-set extensions in a fully automated manner.* Specifically, the goal is to obtain them directly from the high-level language description of the application.

In the following section, we discuss some previous work in the domain; we anticipate our specific goals and contribution in Section 3. We formalise the problem which we try to solve in Section 4. Section 5 introduces our algorithms. Results are described in the two following sections: in Section 6 we detail the experimental setup used and in Section 7 we discuss the results. The paper concludes with some considerations on future directions opened by this work.

2. RELATED WORK

Loosely stated, the problem of identifying instruction-set extensions consists in detecting clusters of operations which, when implemented as a single complex instruction, maximise some metric—typically performance. Such clusters must invariably satisfy some constraint; for instance, they must produce a single result or use not more than four input values. We will formalise the identification and selection problem that our algorithm solves in Section 4, but use this generic formulation to discuss related work.

A recent example of synthesis of application-specific instructions can be found in [5]: the goal is to add special single- and multiple-cycle instructions to a small set of primitive instructions. The authors essentially concentrate on a selection problem which targets a maximal reuse of complex instructions and a minimal number of instructions selected. The reuse goal is likely to favour the identification of small clusters of primitive operations; hence, heuristically, the authors prune the search space by explicitly limiting the
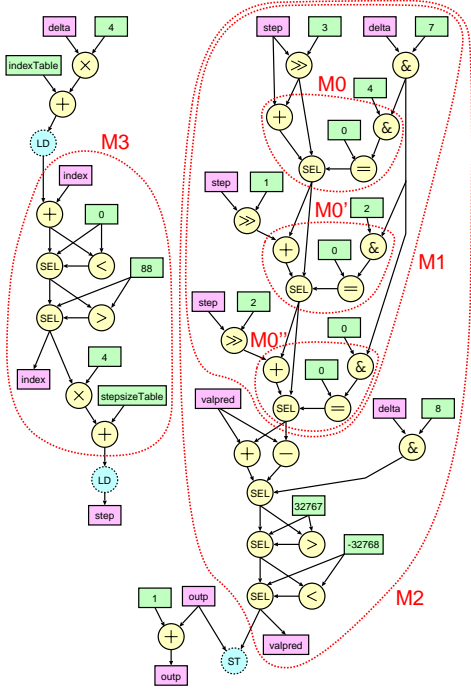
Figure 1: Motivational example from the *adpcmde-code* benchmark [13]. SEL represents a selector node and results from applying an if-conversion pass to the code.

complexity of the special instructions. Our philosophy is different and we directly formulate as our goal to achieve a maximal gain per special instruction.

In other works such as [11] or [2], the authors use approaches combining template matching (*instruction selection*, as it is called in compilers) and template generation (*identification*, in our parlance) for ASIPs. The main specificity of the approach described in [11] is that clustering is based on the frequency of node types successions—e.g., multiplications followed by additions—rather than of frequency of execution of specific nodes. The emphasis on recurrent patterns somehow relates this work to [5]: the authors observe that the number of operations per cluster is typically small and conclude that simple pairs of operations appear the best candidates. Their work does not account for constraints on the number of inputs and outputs of the clusters. The work in [2] is very similar from the identification perspective, although the overall goal and architectural context is rather different.

Work in reconfigurable computing is often more in line with our goal (e.g., [14], [1], [12], [18]). Yet, identification algorithms are relatively simple and almost invariably target clusters producing a single result. Usually, clusters or subgraphs are somehow grown from their output nodes by adding predecessors until some constraints are violated. More formal approaches such as the one described in [1] guarantee a decomposition in maximal single-output subgraphs: unfortunately, the approach cannot be easily extended to multiple output subgraphs and the property of maximal size does not represent optimality under constraints on the number of inputs.

In [3], the identification problem is addressed in a manner

similar to ours in the context of hardware/software partitioning. A simple clustering algorithm is used, called *clubbing*, to enforce limits on the input and output counts (to 3 and 2 respectively, in the examples) and to ensure deterministic functionality (see Section 4). Our algorithm is more expensive but considers the complete design space. Section 7 shows the superiority of the algorithm presented here with respect to [3] and to [1].

3. MOTIVATION AND CONTRIBUTIONS

Figure 1 shows the dataflow graph of the basic block most frequently executed in a typical embedded processor benchmark. We use this simple but realistic example to motivate our work. The first observation is that identification based on recurrence of clusters would hardly find candidates of more than 3–4 operations. Additionally, one should notice that recurring clusters such as M0 have several inputs and could be often prohibitive. In fact, choosing larger albeit nonrecurrent clusters might ultimately reduce the number of inputs and/or outputs: subgraph M1 satisfies even the most stringent constraints of two operands and one result. An inspection to the original code suggests that this subgraph represents an approximate 16×3-bit multiplication and is therefore the most likely manual choice of a designer even under severe area constraints. Availability of a further input would include also the following accumulation and saturation operations (subgraph M2 in Figure 1). For different reasons, most existing algorithms would bail out before identifying such large—but rather cheap—subgraphs. Furthermore, if additional inputs and outputs are available, one would like to implement *both* M2 and M3 as part of the same instruction—thus exploiting the parallelism of the two disconnected graphs. To our knowledge, our algorithm is the only one described in literature capable of identifying all the above mentioned instructions depending on the given constraints.

More specifically, this work will improve the state-of-the-art in three respects: Firstly, prior work was mostly limited to instructions with a single output (with the exceptions of two outputs in [3] and several outputs only in very specific cases in [18]). Our technique identifies custom instructions with any number of outputs up to a user-specified constraint. Note that current VLIW architectures like ST200 and TMS320 can commit 4 values per cycle and per cluster.

Secondly, only connected subgraphs can be identified by previous techniques (apart again from an exception in [18], where only very particular kinds of disconnected graphs can be found). Instead, the present method can detect any kind of disconnected graphs, which results in the possibility of automatically identifying also SIMD-like instructions.

Lastly, many previous works lack a formal methodology for identification and selection of candidates. Here identification and selection are coupled and solved formally at once.

4. PROBLEM STATEMENT

We call $G(V, E)$ the DAGs representing the dataflow of each basic block; the nodes V represent primitive operations and the edges E represent data dependencies. Each graph G is associated to a graph $G^+ (V \cup V^+, E \cup E^+)$ which contains additional nodes V^+ and edges E^+. The additional nodes V^+ represent input and output variables of the basic block. The additional edges E^+ connect nodes V^+ to V,
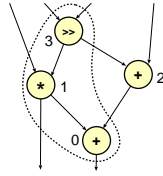
Figure 2: A nonconvex, and thus illegal, subgraph. Numbers refer to topological order explained in Section 5.

Figure 3: The search tree corresponding to the graph shown in Figure 2.

and nodes V to V^+.

A *cut* S is a subgraph of G: $S \subseteq G$. There are $2^{|V|}$ possible *cuts*, where $|V|$ is the number of nodes in G. An arbitrary function $\mathrm{M}(S)$ measures the merit of a cut S. It is the objective function of the optimisation problem introduced below and typically represents an estimation of the speedup achievable by implementing S as a special instruction.

We call $\mathrm{IN}(S)$ the number of predecessor nodes of those edges which enter the cut S from the rest of the graph G^+. They represent the number of input values used by the operations in S. Similarly, $\mathrm{OUT}(S)$ is the number of predecessor nodes in S of edges exiting the cut S. They represent the number of values produced by S and used by other operations, either in G or in other basic blocks.

Finally, we call the cut S *convex* if there exists no path from a node $u \in S$ to another node $v \in S$ which involves a node $w \notin S$. Figure 2 shows an example of nonconvex cut.

Considering each basic block independently, the identification problem can now be formally stated as follows:

PROBLEM 1. *Given a graph G^+, find the cut S which maximises $\mathrm{M}(S)$ under the following constraints:*

1. *$\mathrm{IN}(S) \leq N_{\mathrm{in}}$,*
2. *$\mathrm{OUT}(S) \leq N_{\mathrm{out}}$, and*
3. *S is convex.*

The user-defined values N_{in} and N_{out} indicate the register-file read and write ports, respectively, which can be used by the special instruction. The convexity constraint is a legality check on the cut S and is needed to ensure that a feasible scheduling exists: as Figure 2 shows, if all inputs of an instruction are supposed to be available at issue time and all results are produced at the end of the instruction execution, there is no possible schedule which can respect the dependences of this graph once S is collapsed into a single instruction.

Since we will allow several special instructions from all basic blocks, we will need to find up to N_{instr} cuts which, together, give the maximum advantage. This problem, often referred as *selection*, is often solved nonoptimally by repeatedly solving Problem 1 on all basic blocks and by simply selecting the N_{instr} best ones. Formally, the problem that we want to solve is:

PROBLEM 2. *Given the graphs G_i^+ of all basic blocks, find up to N_{instr} cuts S_j which maximise $\sum_j \mathrm{M}(S_j)$ under the same constraints of Problem 1 for each cut S_j.*

5. IDENTIFICATION ALGORITHMS

We introduce algorithms to solve the above problems in three steps: (1) find the optimal single cut in a single basic block, (2) find an optimal set of nonoverlapping cuts
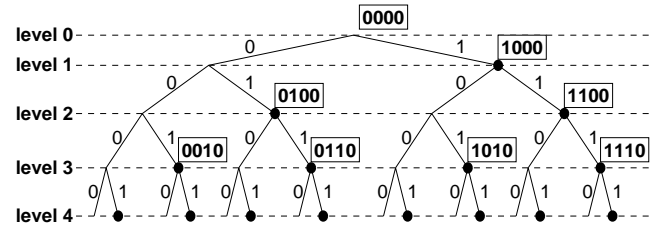
in several basic blocks, and (3) find a near-optimal set of nonoverlapping cuts in several basic blocks.

5.1 Single Cut Identification

Enumerating all possible cuts within a basic block exhaustively is not computationally feasible. We describe here an exact algorithm that explores the complete search space but effectively detects and prunes infeasible regions during the search. The algorithm starts with a topological sort on G. Nodes of G are ordered such that if G contains an edge (u, v) then u appears after v in the ordering. Figure 2 shows a topologically sorted graph. The algorithm uses a recursive search function based on this ordering to explore an abstract search tree.

The search tree is a binary tree of nodes representing possible cuts. It is built from a root representing the empty cut and each couple of 1- and 0-branches at level i represents the addition or not of the node of G having topological order i, to the cut represented by the parent node. Nodes of the search tree immediately following a 0-branch represent the same cut as their parent node, and can be ignored in the search. Figure 3 shows the search tree for the example of Figure 2, with some tree nodes labelled with their cut values. The search proceeds as a preorder traversal of the search tree. It can be shown that in some cases there is no need to branch towards lower levels; therefore the search space is pruned.

Suppose for instance that the output port constraint has already been violated by the cut defined by a certain tree node: adding nodes that appear later in the topological ordering cannot reduce the number of outputs of the cut. Similarly, if the convexity constraint is violated at a certain tree node, there is no way of regaining the feasibility by considering the insertion of nodes of G that appear later in the topological ordering. Considering for instance Figure 2 after inclusion of node 3, the only ways to regain convexity are to either include node 2 or remove from the cut nodes 0 or 3: due to the use of a topological ordering, both solutions are impossible in a search step subsequent to insertion of node 3. As a consequence, when the output-port or the convexity constraints are violated when reaching a certain search tree node, the subtree rooted at that node can be eliminated from the search space.

Figure 4 gives the algorithm in pseudo C notation. The search tree is implemented implicitly, by use of the recursive `search()` function. The parameter `current_choice` defines the direction of the branch, and the parameter `current_index` defines the index of the graph node and the level of the tree on which the branch is taken. When the output port check or the convexity check fails, or when a leaf

```
identification() {
   for (i = 0; i < NODES; i++) cut[i] = 0;
   topological_sort();
   search(1, 0);
   search(0, 0); }

search(current_choice, current_index) {
   cut[current_index] = current_choice;
   if (current_choice == 1) {
      if (!output_port_check()) return;
      if (!convexity_check()) return;
      if (input_port_check()) {
         calculate_speedup();
         update_best_solution(); } }
   if ((current_index + 1) == NODES) return;
   current_index = current_index + 1;
   search(1, current_index);
   search(0, current_index); }
```

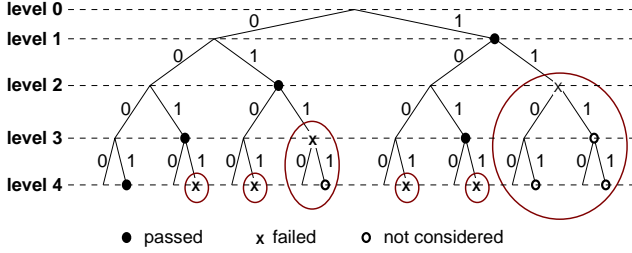**Figure 4: The identification algorithm.**

Figure 5: The execution trace of the algorithm for the graph given in Figure 2 and $N_{\text{out}} = 1$.

Figure 6: Number of cuts considered by the algorithm with $N_{\text{out}} = 2$ and any N_{in}.

Figure 7: A search tree for two cuts.

is reached during the search, the algorithm backtracks. The best solution is updated only if all constraints are satisfied by the current cut.

Figure 5 shows the application of the algorithm to the graph given in Figure 2 with $N_{out} = 1$. Only 5 cuts pass both output port check and the convexity check, while 6 cuts are found to violate either output port constraint or convexity constraint, resulting in elimination of 4 more cuts. Among 16 possible cuts, only 11 are therefore considered.

The graph nodes contain $O(1)$ entries in their adjacency lists on average, since the number of inputs for a graph node is limited in every practical case. Combined with a single node insertion per algorithm step, the `input_port_check`, `output_port_check`, `convexity_check`, and `calculate_speedup` functions can be implemented in $O(1)$ time using appropriate data structures. The overall complexity of the algorithm is therefore $O(2^{|V|})$. Although still exponential, the algorithm reduces in practice the search space very tangibly. Figure 6 shows the run time performance of the algorithm using an output port constraint of two on some basic blocks extracted from several benchmarks. The actual performance is within polynomial bounds in all practical cases considered, however an exponential tendency is also visible. Constraint based subtree elimination plays a key role in the algorithm performance: the tighter the constraints are, the faster the algorithm is.

5.2 Optimal Selection Algorithm

The algorithm described in the previous section can be easily adapted to identify multiple cuts from a single graph. If M is the number of cuts to be identified within a basic block, it suffices to build a similar search tree where every node ma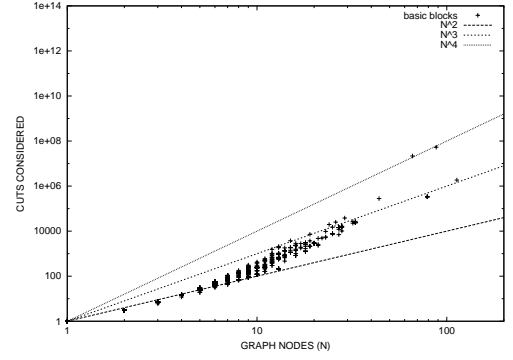kes $M + 1$ branches instead of 2.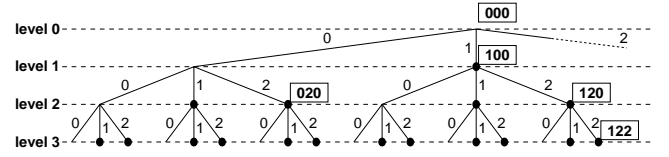 Figure 7 shows a fragment of a tree for $M = 2$. Nodes of the search tree now represent M cuts: an n-branch at level i leads to inclusion of the graph node with index i in the n-th cut.

Our optimal selection algorithm begins by applying the single-cut identification algorithm on each basic block ($M = 1$). The first cut is chosen from the basic block which offers the largest speed-up improvement. Then at each iteration, the algorithm increments the value of M for the basic block which was chosen by the previous iteration, does multiple-cut identification on this basic block with the new value of M, and calculates the improvement. Again, the new cut is chosen from the basic block that gives the largest speed-up improvement. The iterations continue until N_{instr} cuts are chosen. The algorithm can be proven to return optimal solutions by applying the multiple-cut identification algorithm at most $N_{\text{instr}} + N_{\text{bb}} - 1$ times. Figure 8 illustrates the algorithm with a simple example.

5.3 Iterative Selection Algorithm

Repeated calls to the multiple-cut identification algorithm on large basic blocks may result in impracticable computational complexity. To avoid this, we also used a heuristic approach consisting in iterative applications of the single-cut identification algorithm to the same basic block. Previously identified cuts are merged into single graph nodes, and are excluded from forthcoming identification steps. We will compare the results of the two selection strategies.

6. EXPERIMENTAL SETUP

To measure the speedup achieved by our algorithms, we assumed a particular function $M(\cdot)$ to express the merit of a specific cut. $M(S)$ represents an estimation of the speedup achievable by executing the cut S as a single instruction in a specialised datapath.

In software, we estimate the latency in the execution stage of each instruction; in hardware, we evaluate the latency of each operation by synthesising arithmetic and logic oper-
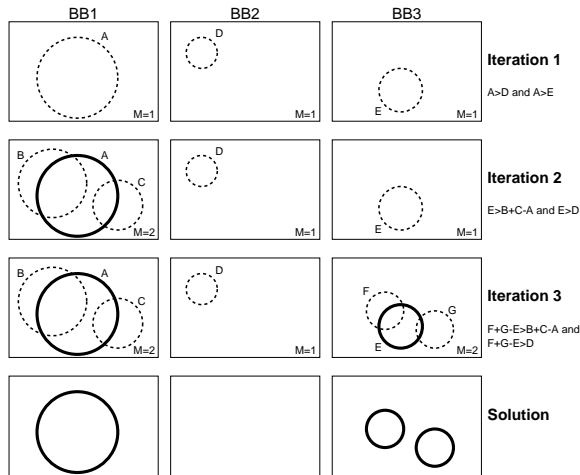
Figure 8: Optimal selection of three cuts in three basic blocks. Circles represent cuts—that is subgraphs, of the basic blocks. Dashed circles are best candidates returned by five calls to the multiple-cut identification algorithm of Section 5.2

ators on a common $0.18\mu m$ CMOS process and normalise to the delay of a 32-bit multiply-accumulate. The accumulated software values of a cut estimate its execution time in a single-issue processor. The latency of a cut as a single instruction is approximated by a number of cycles equal to the ceiling of the sum of hardware latencies over the graph critical path.

The difference between the software and hardware latency is used to estimate the speedup. Although quite rough, this model is also very fast to evaluate and hence adapted for use in the inner loop of our identification algorithm, where by no means one could use a computationally heavier model.

7. RESULTS

The described algorithms were implemented within the MachSUIF framework [15] and tested on a subset of the MediaBench [13] suite benchmarks. Application C-code is compiled to MachSUIF intermediate representation and pre-processed with a classic if-conversion pass.

In order to show the potentials of our algorithms with respect to the state of the art, we have implemented two identification algorithms which are denoted by Clubbing and MaxMISO, and are published respectively in [3] and in [1]. The first is a greedy linear-complexity algorithm that can detect n-input m-output graphs, where n and m are user parameters. The second is a linear complexity algorithm that identifies single-output and unbounded-input graphs.

Figure 9 shows the performance improvement of our algorithms, called Optimal and Iterative (see Sections 5.2 and 5.3, respectively), when compared to Clubbing and MaxMISO, for different benchmarks and for different input and output constraints. The presented results are for up to 16 special instructions.

Four points should be noticed: Firstly, the difference between Optimal and Iterative is usually null and is in all cases irrelevant; we will therefore retain the iterative selection algorithm (note that the Optimal algorithm could not be run on the *adpcmdecode* benchmark due to the size of the basic
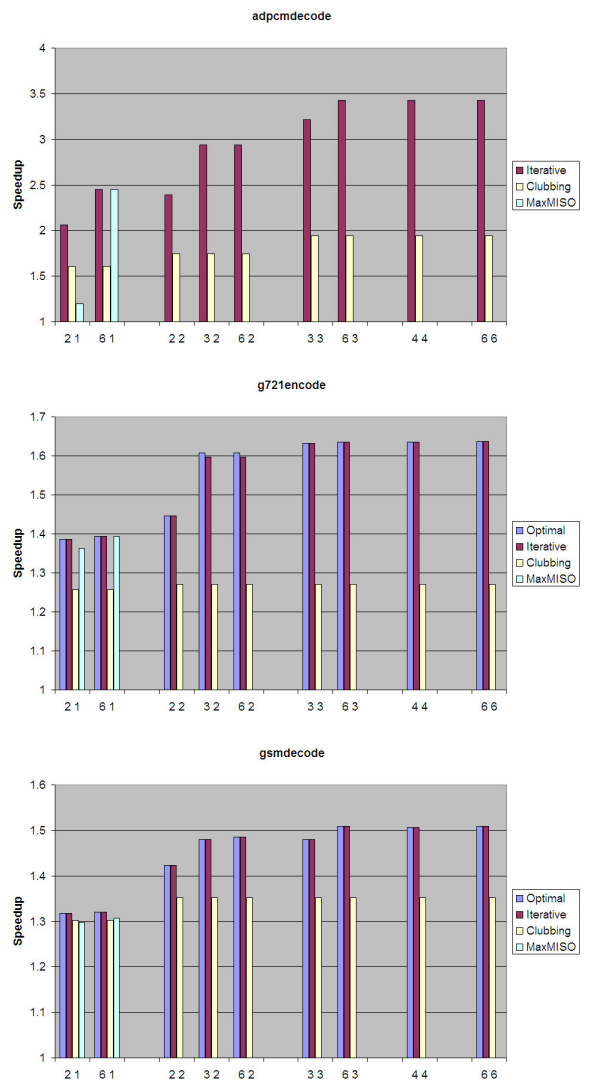
Figure 9: Comparison of estimated speedup for Optimal, Iterative, Clubbing, and MaxMISO on three MediaBench benchmarks, for some selected input and output constraints.

blocks). Secondly, our algorithms generally outperforms the others. Thirdly, in general for low input/output constraints all algorithms have similar performances, but in the case of higher (and yet still very reasonable) constraints Iterative excels. Finally, a large performance improvement potential lays in multiple output and generally disconnected graphs, and the presented algorithms are the first ones to exploit it.

In the light of our motivation, which we expressed with the help of Figure 1, it is useful to analyse the case of *adpcmdecode*: (a) Clubbing is generally limited in the size of the instructions identified. (b) MaxMISO finds the correct solution (corresponding to M2 in the figure) with a constraint of more than two inputs. Yet, when given two input ports, it cannot find M1 because M1 is part of the larger 3-input MaxMISO M2. (3) Iterative manages to increase the speedup further when multiple outputs are available; in such cases, it may choose at once disconnected subgraphs

such as M2+M3. Iterative is the only algorithm that truly adapts to the available microarchitectural constraints.

Of course, the worst-case complexity of our algorithms is higher than that of Clubbing or MaxMISO, but the average complexity is reduced, as Figure 6 shows. In fact, the overall run times of Iterative were quite reasonable in our tests: in all but extreme cases it took only some seconds; only in a couple of cases with loose constraints, run times where in the order of hours.

Finally, note that the area investment needed to implement the special datapaths for the given benchmarks and for the largest chosen graphs was within the area of a couple of multiply accumulators.

8. CONCLUSIONS

This paper has presented algorithms for identifying clusters of dataflow operations to be implemented as application-specific instructions for existing System-on-Chip processors. This task is essential to automate the specialisation of commercial processors. The algorithms take into account microarchitectural constraints and enforces a legality property on the choice. This work is novel with respect to three points: (1) It considers any register-file write port constraint; it is therefore also able to select multiple-output instructions. (2) It is the first to present algorithms to identify generic disconnected graphs. Quantitative results show the importance of the above points. (3) It is the first to formalise identification and selection and solve them together within the same formal framework.

The experiments show that the estimated speedup is raised dramatically when compared with existing state of the art algorithms. The presented algorithms efficiently prune the design space, although still exponential in the worst case. To process very large basic blocks, such as those obtained by applying instruction-level parallelism techniques (e.g., unrolling) to the original code, we plan to build heuristic solutions around the presented identification algorithm. Future work will also address directly the problem of instruction selection under area constraint. Finally, we are planning to use a retargetable compiler to assess precise speedup potentials—especially in VLIW processors where our estimation model is not suitable.

9. REFERENCES

[1] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG based design approach for reconfigurable VLIW processors. In *Proc. of the Design, Automation and Test in Europe Conf. and Exhibition*, pages 778–79, Mar. 1999.

[2] M. Arnold and H. Corporaal. Designing domain specific processors. In *Proc. of the 9th Intl. Workshop on Hardware/Software Codesign*, pages 61–66, Copenhagen, Apr. 2001.

[3] M. Baleani, F. Gennari, Y. Jiang, Y. Pate, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proc. of the 10th Intl. Workshop on Hardware/Software Codesign*, pages 151–56, Estes Park, Colo., May 2002.

[4] F. Campi, R. Canegallo, and R. Guerrieri. IP-reusable 32-bit VLIW Risc core. In *Proc. of the European Solid State Circuits Conf.*, pages 456–59, Villach, Austria, Sept. 2001.

[5] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–14, June 1999.

[6] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, pages 203–13, Vancouver, June 2000.

[7] T. R. Halfhill. ARC Cores encourages "plug-ins". *Microprocessor Report*, 19 June 2000.

[8] T. R. Halfhill. MIPS embraces configurable technology. *Microprocessor Report*, 3 Mar. 2003.

[9] B. K. Holmer. *Automatic Design of Computer Instruction Sets*. Ph.D. thesis, University of California, Berkeley, Calif., 1993.

[10] I.-J. Huang and A. M. Despain. Synthesis of application specific instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-14(6):663–75, June 1995.

[11] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, 7(4), Oct. 2002.

[12] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proc. of the 5th IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 1999.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., Dec. 1997.

[14] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of the 27th Intl. Symp. on Microarchitecture*, pages 172–80, San Jose, Calif., Nov. 1994.

[15] M. D. Smith and G. Holloway. *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*. Harvard University, Cambridge, Mass., 2000.

[16] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proc. of the 7th Intl. Symp. on High-Level Synthesis*, pages 11–16, 1994.

[17] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proc. of the 38th Design Automation Conf.*, pages 184–88, Las Vegas, Nev., June 2001.

[18] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, pages 225–35, Vancouver, June 2000.