International Journal of Parallel Programming, Vol. 31, No. 6, December 2003 (© 2003)

Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints

Kubilay Atasu,[1] Laura Pozzi,[2] and Paolo Ienne[2]

This paper presents a methodology for automatically designing Instruction-Set Extensions in embedded processors. Many commercially available CPUs now

that can define the set of customised functional units most beneficial for a given application are missing. Only a few algorithms exist but are severely limited in the type and size of operation clusters they can choose and hence reduce significantly the effectiveness of specialisation. A more general algorithm is presented here which selects maximal-speedup convex subgraphs of the application data-flow graph under fundamental microarchitectural constraints, and which improves significantly on the state of the art.

KEY WORDS: Customisable processors; instruction-set extensions; hardware/software codesign; automatic partitioning.

1. INTRODUCTION

Two routes can be followed by system-on-chip designers in order to build a specialised processor. The one most in vogue in the last decade 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.[1–3] In that context, the goal is

[1] Computer Engineering Department, Bogazici University, Istanbul, Turkey. E-mail: atasu@cmpe.boun.edu.tr

[2] Processor Architecture Laboratory, Swiss Federal Institute of Technology Lausanne (EPFL), Lausanne, Switzerland. E-mail: {Laura.Pozzi; Paolo.Ienne}@epfl.ch

0885-7458/03/1200-0411/0 © 2003 Plenum Publishing Corporation

typically to design an instruction set which minimises some important metric (e.g., run time, program memory size, execution unit count).

The second route, more recently introduced and more attractive, goes 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[4] and industry.[5–8] 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 specialised processors and describe our target architecture. In Section 3 we present some previous work in the domain; our specific goals and contribution are anticipated in Section 4. We formalise the problem which we try to solve in Section 5. Section 6 introduces our algorithms. Results are described in the two following sections: in Section 7 we detail the experimental setup used and in Section 8 we discuss the results. The paper concludes with some considerations on future directions opened by this work.

2. SPECIALISED PROCESSORS

When a designer is in charge of building a device capable of running a given application, he/she is given a spectrum of choices; part of this spectrum is depicted in Fig. 1. To the left, a dedicated hardware implementation is shown. It exactly matches the application specifications for which it has been designed, and no gap exists which needs to be filled by software; the application is fully implemented in hardware. This represents the best option in terms of speed and power consumption, and the worse in terms of flexibility and time to market. At the other end of the spectrum, a general purpose processor is depicted to the right. Its hardware can be seen as somehow neutral: it does not fit any specific application in particular. However, a software layer—the software implementation of the application—allows the processor to be programmed and to run any task. This solution is the slowest and most power consuming, but the most flexible and fastest to market. Between these two extremes lays the philosophy

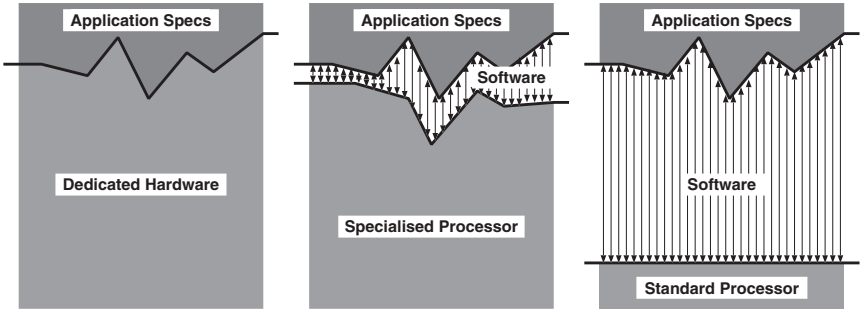

Fig. 1. Specialising a processor might achieve the optimal compromise between speed, power design goals and flexibility.

adopted by many practitioners and in this paper—that of specialised processors. Here, while the hardware does not match precisely the application to serve, the processor has been "shaped" so that it better suits the application goals. This is achieved through addition of *Application-Specific Functional Units (AFUs)* which are used to extend the standard instruction set of the processor. As a consequence, the gap between application and software is partially filled: the AFUs have "shaped" the processor hardware to fit the application specifications. A layer of software can and must still be used in order to program the processor; this retains flexibility, while reaching a good compromise for other design goals such as speed, power consumption, and time to market.

Application-Specific Functional Units are the core of an important class of specialised processors. AFUs can vary in size, in number of ports to and from the register file, in the tightness of their coupling to the rest of the processor, and of course in the functionality they implement. Figure 2 shows a coarse view of a specialised processor organisation. Several input and output ports are provided between AFU and register file, while no direct memory access is possible. In the scope of this paper, the AFU implements an arbitrary combinational function, extracted from the dataflow of the application. It does not contain any architecturally visible state (i.e., registers or memory) and cannot include memory access operations —some extensions are addressed in Section 9. Among the decisions that a specialised processor designer must take, this paper is concerned with the selection of the AFU functionalities—i.e., with the definition of best instruction set extensions compatible with the microarchitecture of the processor at hand. Moreover, the problem is approached in an automated manner, and an algorithmic solution is proposed.

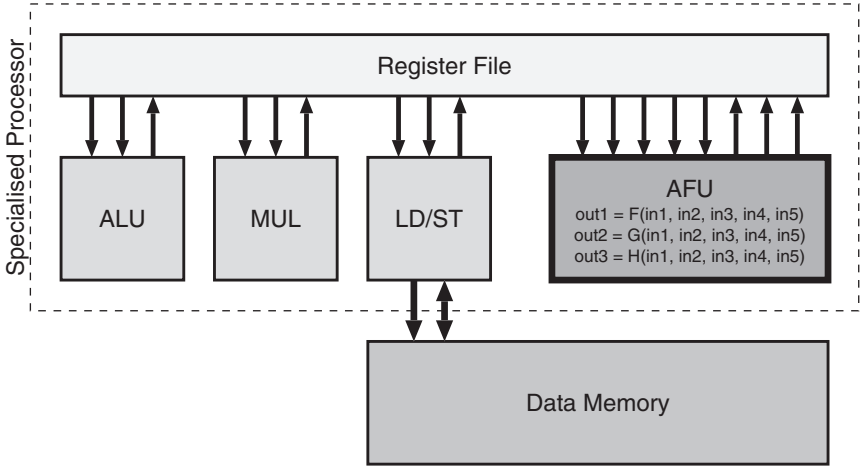

Fig. 2. A processor with a five-input three-output application-specific functional unit.

3. 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 5, but use this generic formulation to discuss related work.

In a recent example of synthesis of application-specific digital signal processors,[9] 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 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,[10, 11] 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 one of these approaches[10] 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 the mentioned one on signal processors:[9] 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 other mentioned work[11] 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.[12–15] 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 algorithms have been presented[13] and 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 a work in the context of hardware/software partitioning,[16] the identification problem is addressed in a manner similar to ours. 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 5). Our algorithm is more expensive but considers the complete design space. Section 8 shows the superiority of the algorithm presented here with respect to two previous techniques.[13, 16]

4. MOTIVATION AND CONTRIBUTIONS

Figure 3 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 16×4-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 Fig. 3). For different reasons, most existing algorithms

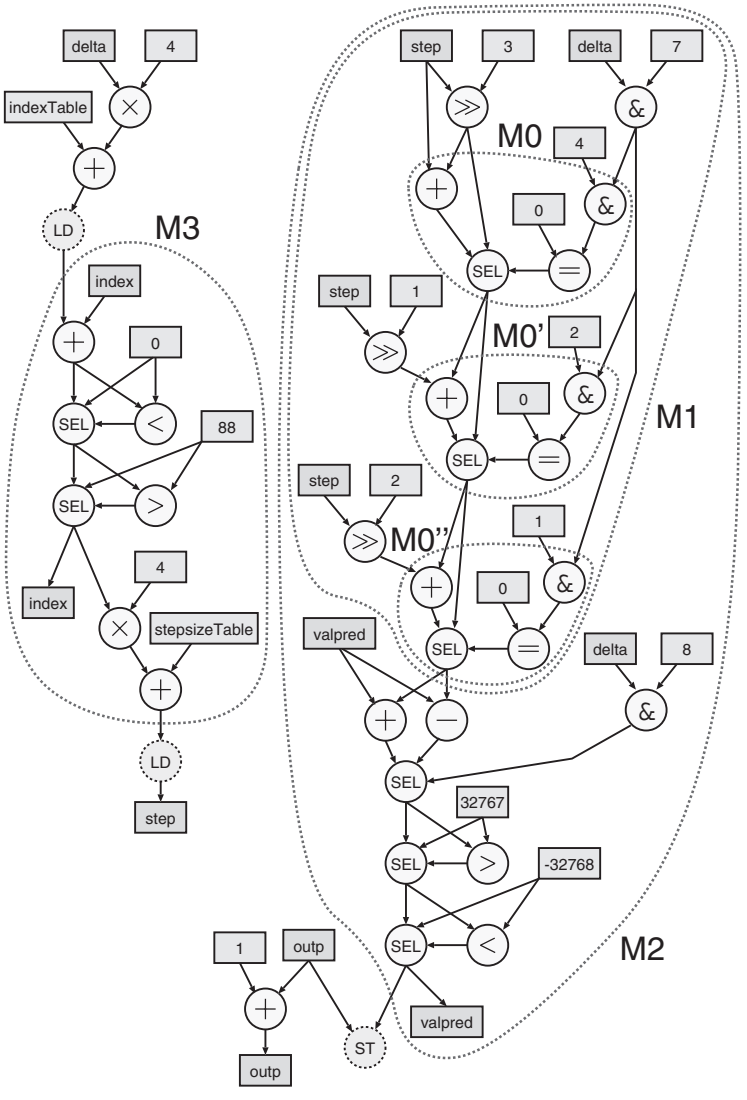

Fig. 3. Motivational example from the *adpcmdecode* benchmark.[17] SEL represents a selector node and results from applying an if-conversion pass to the code.

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[16] and several outputs only in very specific cases[15]). 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 case of very particular disconnected graphs[15]). 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.

5. 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, 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.

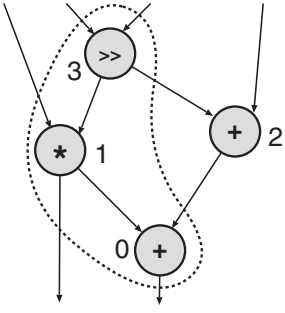

Fig. 4. A nonconvex, and thus illegal, subgraph. Numbers refer to topological order explained in Section 6.

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 4 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. $\mathrm{IN}(S) \leqslant N_{\mathrm{in}}$,
2. $\mathrm{OUT}(S) \leqslant N_{\mathrm{out}}$, 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 can be found by a typical compiler: as Fig. 4 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, referred here 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.

6. 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.

6.1. 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 4 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 5 shows the search tree for the example of Fig. 4, 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

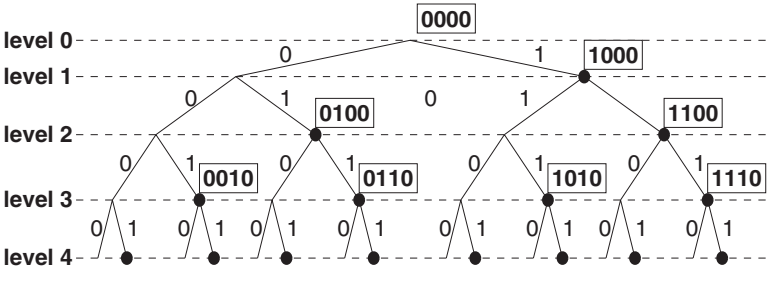

Fig. 5. The search tree corresponding to the graph shown in Fig. 4.

the cut. Similarly, if the convexity constraint is violated at a certain tree node, there is no way of regaining feasibility by considering the insertion of nodes of G that appear later in the topological ordering. Considering for instance Fig. 4 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 6 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 is reached during the search, the algorithm backtracks. The best solution is updated only if all constraints are satisfied by the current cut.

```
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);
}
```

Fig. 6. The identification algorithm.

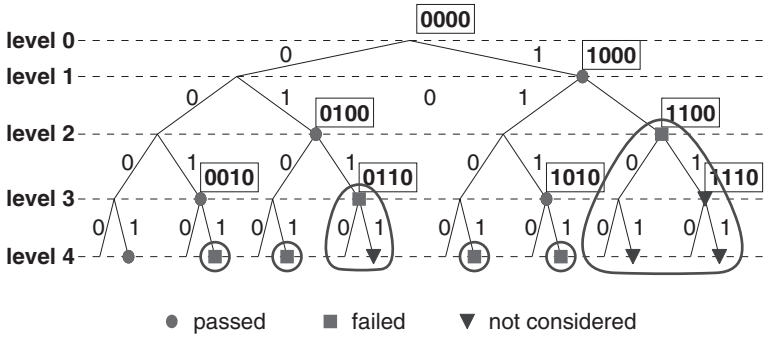

Fig. 7. Execution trace of the algorithm for the graph of Fig. 4 and $N_{out} = 1$.

Figure 7 shows the application of the algorithm to the graph given in Fig. 4 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 in this small example.

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 8 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.

6.2. 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 makes $M+1$ branches instead of two. Figure 9 shows a

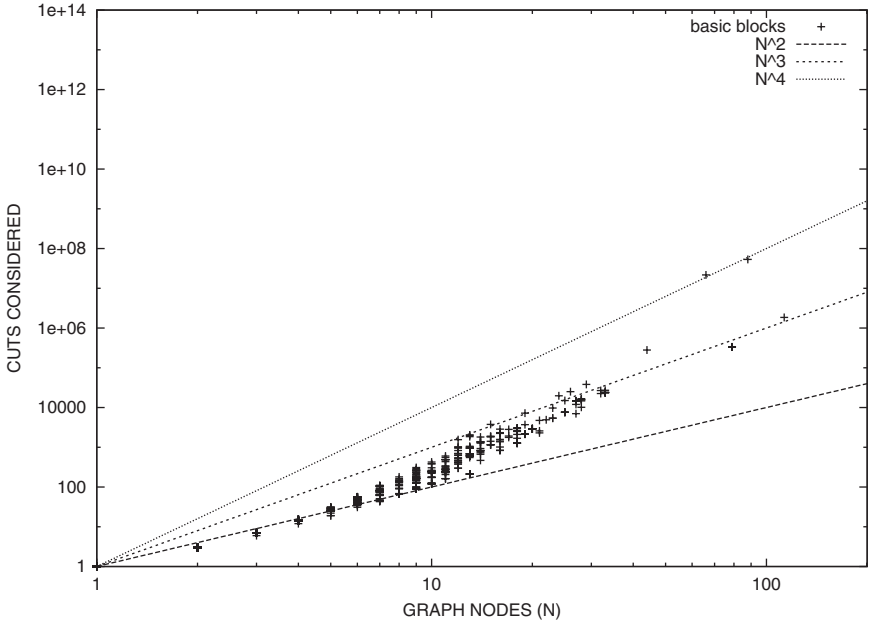

Fig. 8. Number of cuts considered by the algorithm with $N_{out} = 2$ and any N_{in}, for several graphs (basic blocks) with size varying between 2 and around 100 nodes.

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 nth 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

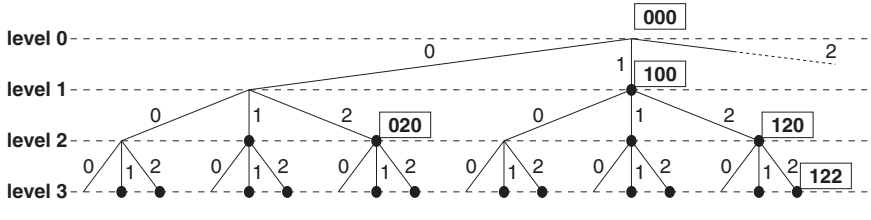

Fig. 9. A search tree for two cuts.

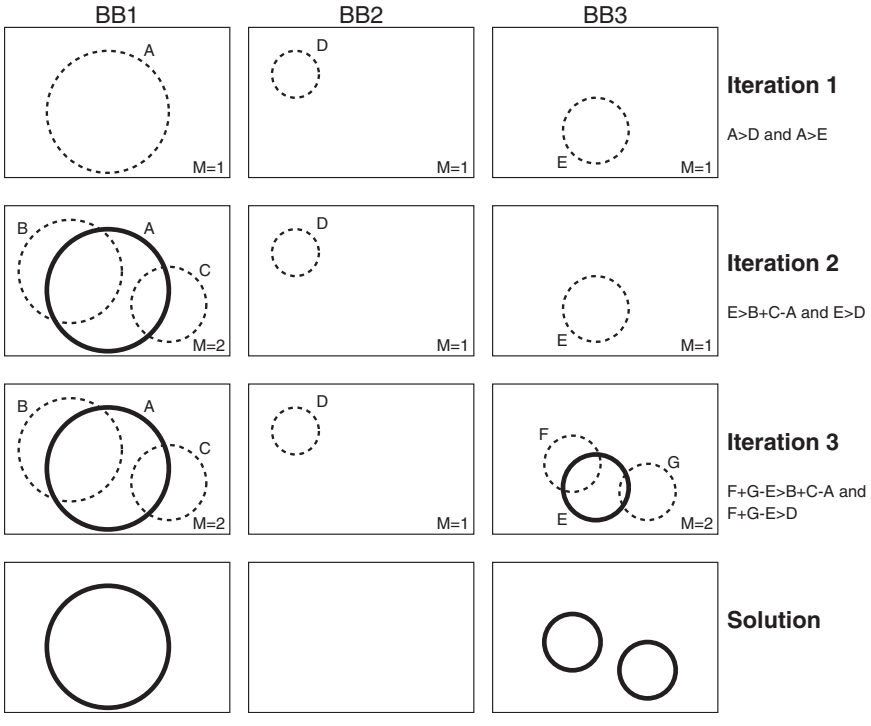

Fig. 10. Optimal selection of three cuts in three basic blocks. Circles represent cuts—that is subgraphs, of the basic blocks. Dashed circles are best candidates returned by five calls to the multiple-cut identification algorithm of Section 6.2.

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 10 illustrates the algorithm with a simple example.

6.3. 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 in Section 8.

7. 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 synthesizing arithmetic and logic operators on a common 0.18 μm 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 apt for use in the inner loop of our identification algorithm, where by no means one could use a computationally heavier model.

8. RESULTS

The described algorithms were implemented within the MachSUIF framework[18] and tested on a subset of the MediaBench[17] suite benchmarks. Application C-code is compiled to MachSUIF intermediate representation and preprocessed 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[16] and MaxMISO.[13] 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 11 shows the performance improvement of our algorithms, called Optimal and Iterative (see Sections 6.2 and 6.3, 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 large size of the basic blocks). Secondly, our algorithms generally outperforms the others. Thirdly, in general for low input/output constraints all algorithms

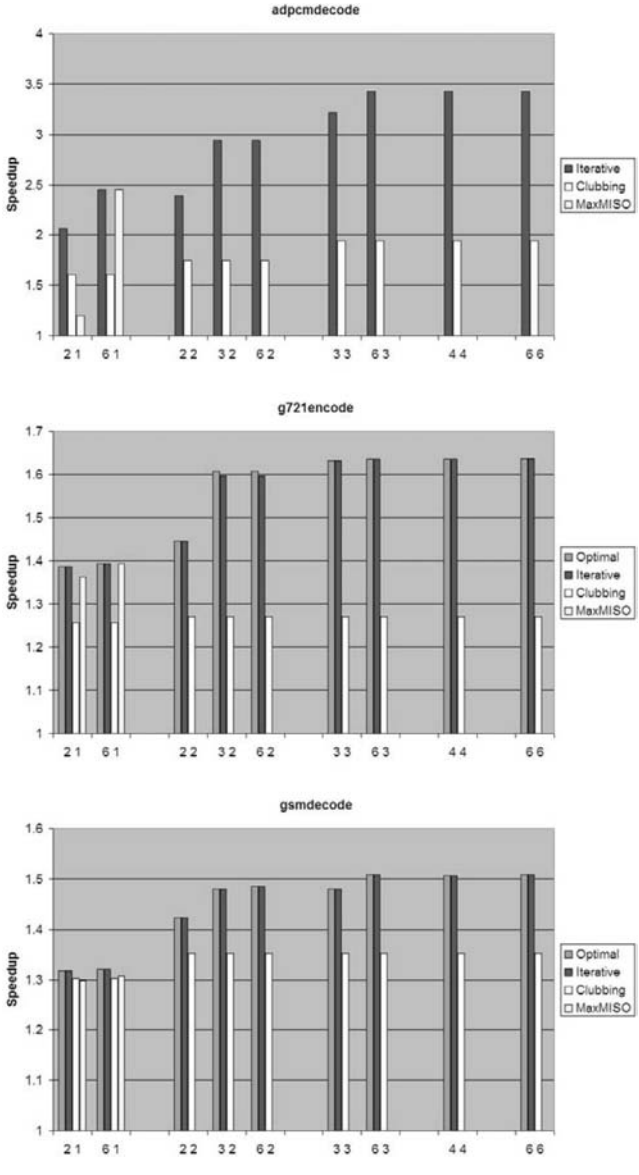

Fig. 11. Comparison of estimated speedup for Optimal, Iterative, Clubbing, and MaxMISO on three MediaBench benchmarks, for some selected input and output constraints.

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 Fig. 3, 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**. (c) 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 much higher than that of Clubbing or MaxMISO, but it is on average well below exponential complexity, as Fig. 8 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.

9. 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 enforce 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 this and the above point. (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, and inclusion of registers and local memories in the AFUs. 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.

ACKNOWLEDGMENTS

This work was performed while Kubilay Atasu was with the Processor Architecture Laboratory, Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland.

REFERENCES

1. B. K. Holmer, *Automatic Design of Computer Instruction Sets*, Ph.D. thesis, University of California, Berkeley, California (1993).
2. J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, Instruction Set Definition and Instruction Selection for ASIPs, *Proceedings of the 7th International Symposium on High-Level Synthesis*, Niagara-on-the-Lake, Ontario, pp. 11–16 (April 1994).
3. 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):663–675 (June 1995).
4. F. Campi, R. Canegallo, and R. Guerrieri, IP-Reusable 32-Bit VLIW Risc Core, *Proceedings of the European Solid State Circuits Conference*, Villach, Austria, pp. 456–459 (September 2001).
5. T. R. Halfhill, MIPS Embraces Configurable Technology, *Microprocessor Report*, (3rd March 2003).
6. A. Wang, E. Killian, D. Maydan, and C. Rowen, Hardware/Software Instruction Set Configurability for System-on-Chip Processors, *Proceedings of the 38th Design Automation Conference*, Las Vegas, Nevada, pp. 184–188 (June 2001).
7. T. R. Halfhill, ARC Cores Encourages "Plug-Ins," *Microprocessor Report* (19th June 2000).
8. P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, Lx: A Technology Platform for Customizable VLIW Embedded Processing, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, pp. 203–213 (June 2000).
9. 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 C*, **48**(6):603–614 (June 1999).
10. R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh, Instruction Generation for Hybrid Reconfigurable Systems, *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, **7**(4):605–627 (October 2002).
11. M. Arnold and H. Corporaal, Designing Domain Specific Processors, *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, Copenhagen, pp. 61–66 (April 2001).

12. R. Razdan and M. D. Smith, A High-Performance Microarchitecture with Hardware-Programmable Functional Units, *Proceedings of the 27th International Symposium on Microarchitecture*, San Jose, California, pp. 172–180 (November 1994).
13. C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, A DAG Based Design Approach for Reconfigurable VLIW Processors, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 778–779 (March 1999).
14. B. Kastrup, A. Bink, and J. Hoogerbrugge, ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator, *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, California (April 1999).
15. Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, pp. 225–235 (June 2000).
16. M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli, HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform, *Proceedings of the 10th International Workshop on Hardware/Software Codesign*, Estes Park, Colorado, pp. 151–156 (May 2002).
17. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, pp. 330–335 (December 1997).
18. M. D. Smith and G. Holloway, *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*, Harvard University, Cambridge, Massachusetts (2000). http://www.eecs.harvard.edu/hube/software/.