Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints

Ajay Kumar Verma
Computer Science & Engineering
Indian Institute of Technology
Kanpur, India
akverma@cse.iitk.ac.in

Kubilay Atasu
Miljan Vuletić
Laura Pozzi
Paolo Ienne
Processor Architecture Laboratory
Swiss Federal Institute
of Technology Lausanne (EPFL)
Lausanne, Switzerland
Kubilay.Atasu@epfl.ch
Miljan.Vuletic@epfl.ch
Laura.Pozzi@epfl.ch
Paolo.Ienne@epfl.ch

ABSTRACT

Embedded processors for Systems-on-Chip offer unique possibilities to increase the performance under tight cost budgets. One such possibility, recently offered by many commercial processors, is the extension of the instruction set for a specific application—that is, the introduction of customised functional units. It is essential, both to limit cost and risk in the design process and to explore appropriately the design space, to develop algorithms that decide automatically from high-level application code which operations are to be carried out in customised functional units. A few algorithms exists 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. We show that our algorithm is an essential component for the successful exploitation of the specialisation potentials and that the hardware cost incurred to specialise the processor is moderate when compared to its effectiveness.

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 ([11], [9], [18], [10]). In that context, the goal is typically to cluster all the atomic operations required for an application into an instruction set which minimises some important metric (e.g., execution time, program memory size, number of execution units).

More recently, the attention has shifted toward extending generic processors with units specialised for a given domain, rather than designing completely custom processors. These additional functional units or configurable datapaths can be thought as tightly-coupled fine-grained coprocessors. 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., [14], [4]) and industry (e.g., [19], [8], [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 (possibly including its implementation as a hard-macro) and its extensible toolset can be leveraged: design efforts must focus exclusively on the special datapath.

Typically, it is an expert developer who is charged of defining the operations implemented in these special functional units (e.g., [19]). 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 algorithm. Results are described in the two following sections: in Section 6 we detail the experimental setup we have 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 (that is, subgraphs of the dataflow graph) 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, use not more than four input values, and/or require not more than a given area for the corresponding datapath. We will formalise the identification 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 complexity of the special instructions. Although our work is complementary to theirs (that is, their selection process can be applied to the clusters which we identify), our philosophy is different and we directly formulate as our goal to achieve a maximal gain per special instruction.

In other works such as [12] 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 [12] 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., [16], [1], [13], [20]). 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 and directly ranks the solutions in function of the estimated speedup.

3. MAIN CONTRIBUTIONS

We consider the *Directed Acyclic Graphs* (DAGs) which represent the application *basic blocks*. Figure 1 shows the dataflow graph of a basic block producing five results from ten input values. Assuming this basic block to be among those most frequently executed, one would expect to map as much as possible of it onto a specialised datapath. Existing identification algorithms would mostly concentrate on subgraph `M2` which has a single output. Conversely, one would like to (1) allow multiple outputs, and (2) consider disconnected graphs: if, for example, a 6 input 4 output constraint is given, subgraph `M1` should be identified. Additionally, even under a single output constraint, subgraph `M3` might be a better choice despite containing one node less than subgraph `M2`: what counts is the speedup that nodes can bring when implemented in a custom datapath, and therefore simple shifts and bitwise operations should be preferred because they can bring most advantage when collapsed together in a single instruction.

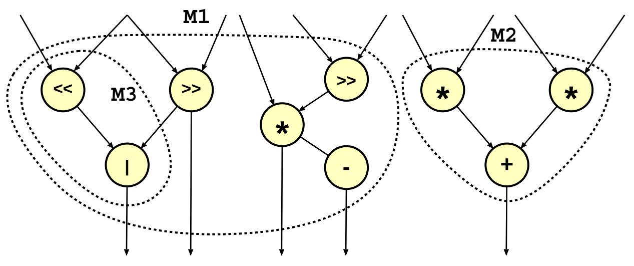


Figure 1: Motivational example for this work. By allowing only largest single-output connected graphs, only subgraph `M2`, to the right, can be identified.

Hence, this work will improve the state-of-the-art in three respects: Firstly, we identify subgraphs for the specialised datapaths not relying *only* on topological features. Instead, we consider an estimated potential speedup of each subgraph and our identification algorithm detects clusters with highest estimated speedup, not being limited by any topological feature except those needed for correct functioning and correct integration in the microarchitecture. We do not care for frequency of appearance of the subgraph but look for maximal estimated advantage.

Secondly, prior work was mostly limited to instructions with a single output—only some exceptions exist ([20], [2], [3]) with various restrictions. As current VLIW architectures show, there is no reason to limit the results which can be committed in a cycle to one or two (e.g., ST200 and TMS320 can commit 4 values per cycle and per cluster). Our technique identifies custom instructions with any number of outputs up to a user-specified constraint.

Thirdly, since most previous techniques work by grouping adjacent nodes, only connected subgraphs can be identified. Instead, the present method can detect disconnected graphs, which results in the possibility of automatically identifying also SIMD-like instructions. Note that one exception does exist in the literature ([20]) where SIMD-like multiple-output disconnected graphs are identified; yet, the method described there addresses only particular subgraphs and does not apply to the general case.

4. PROBLEM STATEMENT

We call $G(V, E)$ the DAG representing the dataflow of a basic block; the nodes V represent primitive operations and the edges E represent data dependencies. Each graph G is associated to a graph $G^+\left(V \cup V^+, E \cup E^+\right)$ which contains additional nodes V^+ and edges E^+. The additional nodes V^+ represent input variables to the basic block and results

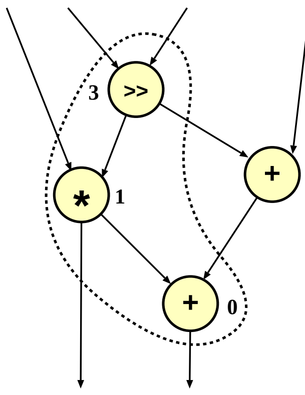


Figure 2: A non-convex, and thus illegal, subgraph which cannot be implemented in a special instruction. Numbers refer to topological order explained in Section 5.

of the basic block used elsewhere in the program. 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. Section 6 discusses a possible function $\mathrm{M}(\cdot)$ approximating the speedup for typical single-issue RISC processors.

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.

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 instructions 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.

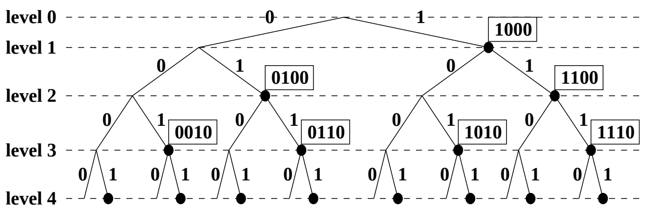


Figure 3: The search tree corresponding to the graph shown in Figure 2.

5. IDENTIFICATION ALGORITHM

Enumerating all possible cuts exhaustively is not feasible in terms of computational complexity. We describe here an algorithm that finds *exact solutions* with subexponential average-case time complexity. The algorithm is able to avoid considering some parts of the search space in specific cases.

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 can be 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. In fact, the only way to decrease the output port requirements is to include further nodes that have a data dependency on the nodes already in the cut. Therefore, such nodes appear earlier in the topological ordering and thus will not be considered anymore. 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 is 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 subsequent search steps following 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 the 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

```
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.

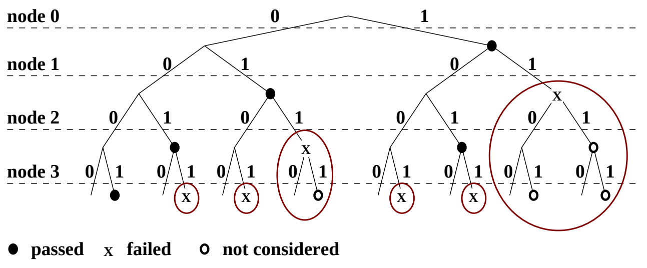


Figure 5: The execution trace of the algorithm for the graph given in Figure 2 and $N_{\text{out}} = 1$.

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 the constraints are satisfied by the current cut.

The complexity of the input-port and output-port checks is $O\left(\left|E \cup E^{+}\right| + \left|V \cup V^{+}\right|\right)$ since all the nodes must be visited once and in the worst case all of the edges need to be visited once too. The convexity check can be done by searching for paths that start and end with nodes that are included in the cut, but have at least one node in between that is not included. If such a path is found the check fails. The convexity check can be implemented in $O\left(|E| + |V|\right)$ time by applying a depth first search on G. In the worst case the identification algorithm considers all useful $2^{|V|} - 1$ cuts, which implies that the worst case complexity of the algorithm is $O\left(2^{|V|} \cdot \left(\left|E \cup E^{+}\right| + \left|V \cup V^{+}\right|\right)\right)$.

Figure 5 shows 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.

A dual algorithm can be developed reversing the topological ordering and using the input port constraint. Since the search space reduction is most effective for tight constraints, either the input-based or output-based algorithm can be used depending on which constraint is tighter. Quantitative results on the reduction of the search space in practical

Operator	Precision in bits	Relative Delay Λ_{hw}
MAC	32 x 32 + 64	1.00
Adder	32 + 32	0.25
Divider	32 / 32	9.61
Barrel shifter	32	0.16
Bitwise AND/OR	any	0.02

Table 1: Examples of hardware timing models of some operators.

cases are given in Section 7.

6. EXPERIMENTAL SETUP

For our experiments, we assumed a particular function $\mathrm{M}(\cdot)$ to express the merit of a specific cut. $\mathrm{M}(S)$ represents a rough estimation of the the speedup achievable by executing the cut S as a single instruction in a specialised datapath.

To define $\mathrm{M}(\cdot)$, we attach two values to each node of G: Λ_{sw} represents the estimated latency of the execution phase in the processor pipeline. Λ_{hw} represents the latency of the corresponding operator as a fragment of specialised datapath in hardware. Software latencies are generally 1 for each primitive instruction, with few exceptions such as division. This somehow implies that the processor is single-issue and pipelined, and each instruction occupies the execute stage for a single clock cycle. Dependences between successive instructions are supposed to be corrected by FU-to-FU forwarding paths. This also assumes that the processor has no cache, neither for instructions nor data (as it is the case for many embedded processors) or, equivalently, that caches have perfect hit rates (which is possibly true for tight high-frequency loops—the natural focus of this type of work); similarly loads and stores also take one cycle but are not considered for inclusion in a cut.

Hardware latencies have been calculated by synthesising arithmetic and logic operators on a common $0.18\mu m$ CMOS process with standard cells from a popular library. All operations are considered relatively to the delay of a 32-bit multiply-accumulate (MAC), which is assumed to be the cycle-time limiting factor of the unaugmented processor. Table 1 shows the relative delay of some operators.

These values make it possible to approximate the total time to execute a cut in software (T_{sw}^{S}) or as a special instruction (T_{hw}^{S}):

$$
\begin{aligned}
T_{\text{sw}}^{S} &= \sum_{i \in S} \left(\Lambda_{\text{sw}}\right)_i , \\
T_{\text{hw}}^{S} &= \left\lceil \sum_{i \in \mathrm{CP}(S)} \left(\Lambda_{\text{hw}}\right)_i \right\rceil .
\end{aligned}
$$

In software, the sum is extended over all primitive instructions in the cut (corresponding to the single-issue pipeline assumption). Conversely, in hardware the sum is extended only to the nodes laying on the cut's critical path $\mathrm{CP}(S)$ and it is rounded up to an integral number of cycles. A special instruction can therefore take multiple cycles to execute and its datapath might be pipelined.

Finally, the estimated speedup can be computed as

$$
\mathrm{M}(S) = \frac{T_{\text{tot}}}{T_{\text{tot}} - n_S \left(T_{\text{sw}}^{S} - T_{\text{hw}}^{S}\right)},
$$

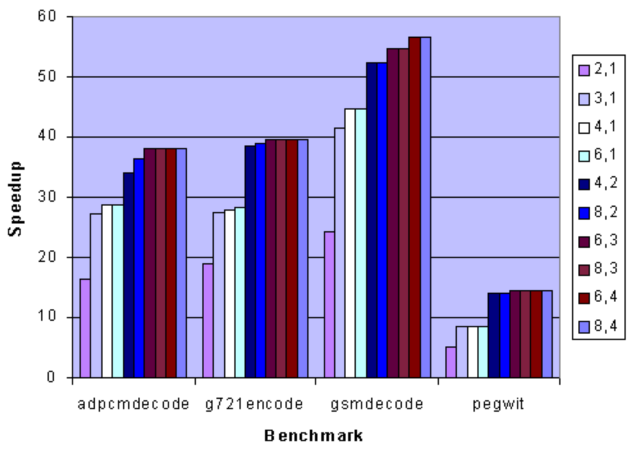


Figure 6: Estimation of speedup achieved by the pruning algorithm for the subset of MediaBench, under different input and output constraints.

where T_{tot} is the total time to run the application and n_S is the number of times the basic block containing S (and hence S itself) is executed. One should notice that the model we use is for estimation only—a fast and convenient way to assess potential special-instruction benefits. For finer decision support, a compiler would be required.

Additionally, to obtain a quantitative estimate of the area needed for implementing a given special instruction, we use information on the area of each operation, implemented in the same ASIC technology described above. Arbitrarily, the area is measured relatively to the MAC unit and the total area for a cut is calculated as the sum of the areas of all its nodes, ignoring possible logic and arithmetic optimisations, on one side, and wiring, on the other.

7. RESULTS

The described algorithm was implemented within the MachSUIF framework [17] and tested on a subset of the MediaBench [15] suite benchmarks. Application C-code is compiled to MachSUIF intermediate representation and the dataflow graphs are processed by our algorithm.

Figure 8 shows the algorithm execution performance in a realistic case. It indicates the number of cuts considered versus all possible cuts, for dataflow graphs corresponding to the basic blocks of two benchmarks and under an input port constraint of 4. For large graphs we observe a reduction of the number of cuts to be explored by several orders of magnitude. The largest basic block, corresponding to a dataflow graph of size 28, required about 5 seconds on a Sun workstation. Under tighter constraints the algorithm runs faster and in practice we were able to process graphs of size up to 50–60 nodes under realistic constraints and in reasonable time.

Next, the estimated speedup achieved with the instructions selected by the algorithm is shown in Figure 6. (In all figures, the indicated speedup is the estimated percentage of saved cycles, and pairs m, n indicate input and output constraints respectively.) Notice that since our algorithm addresses the complete design space, it performs on single-output cases better than or equal to classic single-output

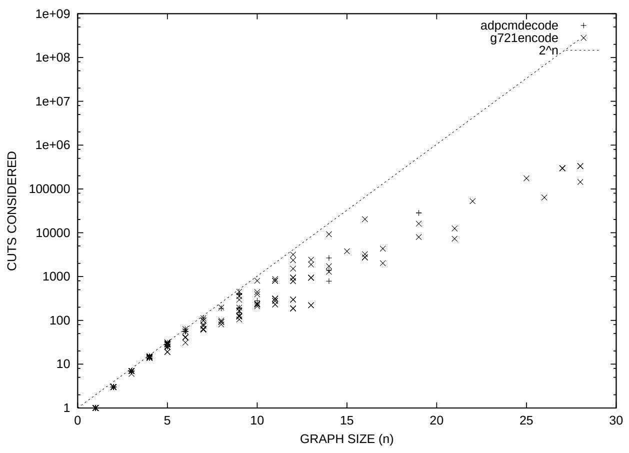


Figure 8: Number of actual cuts analysed on two benchmarks, with $N_{\text{in}} = 4$ and any N_{out}.

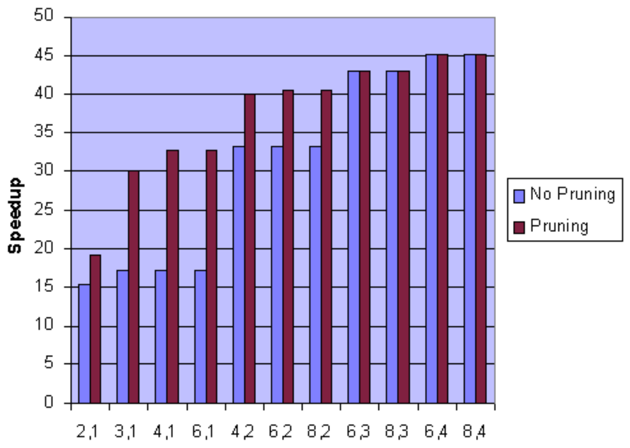
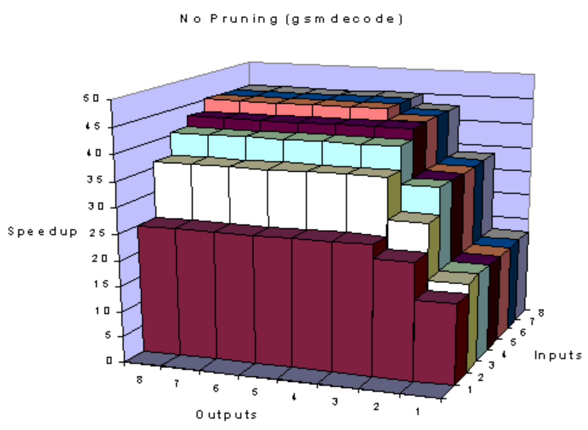


Figure 9: Estimation of speedup advantage brought by the pruning algorithm for the *gsmdecode* benchmark, compared with no-pruning case for some selected constraints. The presented results are for up to 16 special instructions.

identification techniques. Our relaxing of the single-output constraint brings usually more than 10–15% in speedup improvement. This increase represents a performance step otherwise not attainable with single-output identification algorithms, as shown in the figure.

Figure 7 shows the three dimensional surfaces of the estimated speedup for the *gsmdecode* benchmark as a function of input and output constraints. The two graphs compare our algorithm (on the right) to the simple strategy of choosing only basic blocks which respect the given I/O constraint. With this simpler strategy, basic blocks whose inputs or outputs exceed the imposed constraint are rejected without attempting to identify an acceptable cut. We call this strategy *no-pruning*. The surface at the left shows that the speedup in the first graph drops more quickly at the edges than it does in the right one: the no-pruning strategy cannot find enough basic blocks below the constraint (therefore achieves only low speedup for realistically tight constraints),



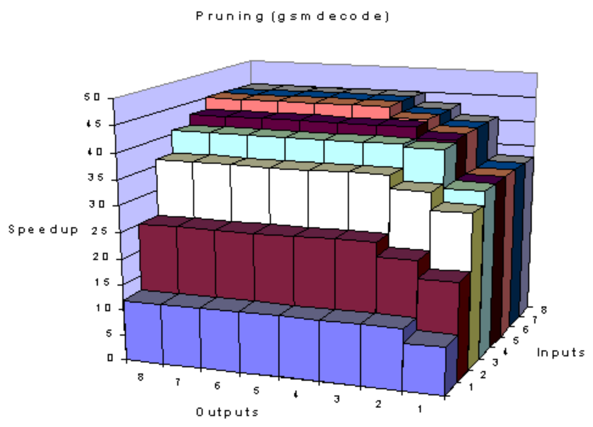


Figure 7: Estimation of speedup advantage on the input/output design space brought by the pruning algorithm for the *gsmdecode* benchmark, contrasted to the no-pruning strategy. The presented results are for up to 16 special instructions.

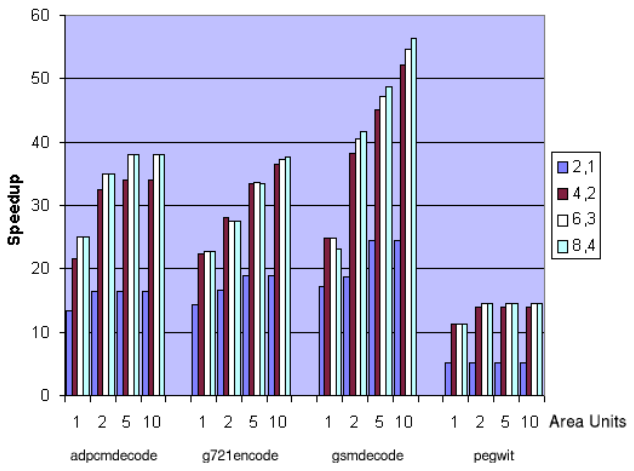


Figure 10: Estimation of area investment in the FUs needed to increase the speedup. The analysis is performed for all of the considered benchmarks and for several constraints on the special instructions I/O and total area. An area unit is the area of 32-bit MAC.

while our algorithm can prune large basic blocks, therefore exploiting them at least partially. For more relaxed constraints (large number of inputs and outputs), the surfaces are, of course, converging to the same plateau. The different speedup achieved for some important points of this graph can be better analysed in Figure 9: for low-constrained points such as (4, 1) or (6, 1) the potential speedup is approximately doubled thanks to our proposed algorithm (e.g., for (4, 1) from 17% to 33%).

Finally, the area investment needed to implement the special datapaths is analysed in Figure 10. It presents the relation of speedup increase to the available area budget. The area has been measured as a ratio to the area of 32-bit MAC operators (beware that no register file size is accounted for). Of course, increasing the area budget provides more speedup; yet, moderate silicon real-estate already bring a tangible speedup, often close to the asymptotic limit.

8. CONCLUSIONS

This paper has presented an algorithm 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 algorithm takes into account register-file port constraints and enforces a legality property on the choice. This work is novel with respect to three points: (1) It is based on a metric to estimate the performance gain of the potential instruction; hence, speedup information alone and not topological relations between nodes are used to choose the DAG partition. (2) It distinguishes from previous work in taking into account any register-file write port constraint; it is therefore also able to select multiple-output instructions. (3) Finally, thanks to the above features, it is also the first algorithm to identify generic disconnected graphs including SIMD-like operations.

The experiments show that the estimated speedup is raised dramatically when application-specific instructions are allowed to produce multiple results—even in the most economically realistic situations with 2–3 write ports. With this algorithm, microarchitecture designers can better weigh the decision of adding new ports to the register-file against their real advantage: for instance, multiple write ports quickly bring the estimated speedup sensibly closer to the asymptotic value for infinite write ports. The silicon area used for automatically identified instruction-set extensions is usually small: in most cases, at least 30% of speedup can be achieved with the approximate area-equivalent of a couple of 32-bit MACs.

The presented algorithm efficiently prunes the design space but is still exponential in the worst case. To process very large basic blocks, such as those obtained by applying to the original code instruction-level parallelism techniques (e.g., unrolling or predication), we plan to build heuristic solutions around the presented algorithm. Future work will also address directly the problem of instruction selection under area constraint—the approach used to obtain some of the present results being elementary. Finally, we are planning to use a retargetable compiler to assess precise speedup potentials—especially in VLIW processors where our estimation model is unadapted.

9. REFERENCES

[1] Cesare Alippi, William Fornaciari, Laura Pozzi, and Mariagiovanna Sami. A DAG based design approach for reconfigurable VLIW processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 778–79, March 1999.

[2] Marnix Arnold and Henk Corporaal. Designing domain specific processors. In *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, pages 61–66, Copenhagen, April 2001.

[3] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Pate, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign*, pages 151–56, Estes Park, Colo., May 2002.

[4] Fabio Campi, Roberto Canegallo, and Roberto Guerrieri. IP-reusable 32-bit VLIW Risc core. In *Proceedings of the European Solid State Circuits Conference*, pages 456–59, Villach, Austria, September 2001.

[5] Hoon Choi, Jong-Sun Kim, Chi-Won Yoon, In-Cheol Park, Seung Ho Hwang, and Chong-Min Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–14, June 1999.

[6] Jennifer Eyre and Jeff Bier. Infineon targets 3G with Carmel2000. *Microprocessor Report*, 17 July 2000.

[7] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 203–13, Vancouver, June 2000.

[8] Tom R. Halfhill. ARC Cores encourages “plug-ins”. *Microprocessor Report*, 19 June 2000.

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

[10] Ing-Jer Huang and Alvin 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] Masaharu Imai, Alauddin Alomary, Jun Sato, and Nobuyuki Hikichi. An integer programming approach to instruction implementation method selection problem. In *Proceedings of the European Design Automation Conference*, pages 106–11, Hamburg, September 1992.

[12] Ryan Kastner, Adam Kaplan, Seda Ogrenci Memik, and Elaheh Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, 7(4), October 2002.

[13] Bernardo Kastrup, Arjan Bink, and Jan Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., April 1999.

[14] Alberto La Rosa, Luciano Lavagno, and Claudio Passerone. A software development tool chain for a reconfigurable processor. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 93–98, Atlanta, Ga., November 2001.

[15] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., December 1997.

[16] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 172–80, San Jose, Calif., November 1994.

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

[18] Johan Van Praet, Gert Goossens, Dirk Lanneer, and Hugo De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, pages 11–16, 1994.

[19] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Design Automation Conference*, pages 184–88, Las Vegas, Nev., June 2001.

[20] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–35, Vancouver, June 2000.