Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets

Laura Pozzi, *Member, IEEE*, Kubilay Atasu, *Student Member, IEEE*, and Paolo Ienne, *Member, IEEE*

***Abstract*—In embedded computing, cost, power, and performance constraints call for the design of specialized processors, rather than for the use of the existing off-the-shelf solutions. While the design of these application-specific CPUs could be tackled from scratch, a cheaper and more effective option is that of extending the existing processors and toolchains. Extensibility is indeed a feature now offered in real designs, e.g., by processors such as Tensilica Xtensa [T. R. Halfhill, *Microprocess Rep.*, 2003], ARC ARCtangent [T. R. Halfhill, *Microprocess Rep.*, 2000], STMicroelectronics ST200 [P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, *Proc. 27th Annu. Int. Symp. Computer Architecture*, 2000, p. 203], and MIPS CorExtend [T. R. Halfhill, *Microprocess Rep.*, 2003]. While all these processors provide development environments with simulation capabilities for evaluating efficiently hand-crafted solutions, the tools to identify automatically the best processor configuration for a given application are less common. In particular, solutions to choose specialized instruction-set extensions (ISEs) have been investigated in the past years but are still seldom part of commercial toolchains. This paper provides a formal methodology and a set of algorithms that help address the problem. It proposes exact algorithms to derive optimal ISEs; exact identification of a single ISE is applicable to basic blocks of up to 1500 assembler-like instructions. This paper also introduces approximate methods that can process basic blocks of larger size. Results show that the described algorithms find solutions close to those that a designer would obtain by a detailed study of the application code. Both heuristic and exact algorithms find ISEs able to speed up unextended processors up to 5.0x. State-of-the-art comparisons show that the presented algorithms outperform existing ones by up to 2.6x.**

***Index Terms*—Application-specific microprocessors, computer instructions, customizable microprocessors, extensible microprocessors, instruction-set extensions, reduced instruction set computing, tightly-coupled coprocessors.**

I. Introduction

In THE COMPUTING world, CPUs have evolved during the last decades from complex instruction set computer (CISC) to reduced instruction set computer (RISC) architectures, and this shift has enabled the introduction of very-high-performance multiple-issue processors. Yet, traditional RISCs, in general computing applications, have not displaced the most common CISC architecture: Many economic and practical issues favored and still favor the market domination by a single architecture—a RISC at heart under the appearance of an old CISC.

This dominance of a single architecture in the general computing world is not at all typical of embedded processing, especially in the area of processors for application-specific integrated circuits (ASIC). Most of the reasons, which favored the concentration on a single architecture in traditional computing, have little or no weight in the ASIC context. For instance, binary compatibility is of less importance since software is provided only for the specific product the ASIC is made for—and often it is completely invisible to the end user. Also, the precise tradeoff between performance, area cost, and power consumption varies widely in the embedded processor world: Typically, it is not absolute maximum performance that matters, but minimum cost or power consumption for a given lower bound on performance; such more precise optimizations are only possible through a wide choice of architectures or customization of existing ones. Finally, in a programmable ASIC, it is economically conceivable to use a nonstandard processor because the latter is integrated and manufactured on the same die with the rest of the application-specific system—of course, in practice, one cannot develop a new processor and its toolchain from scratch for each product.

In this scenario of emerging extensible CPUs, there is a need for techniques that help the designer in deciding how best to extend the base architecture to meet the design goals. While the potential offered by these new machines is noteworthy, what is still needed is a set of design automation tools that start from the software description of an application and lead the architect toward an extended design. In particular, a missing part of the typical design toolchain is that of automatically deciding the instruction-set extensions (ISEs) of the new architecture (a recent commercial exception is discussed in Section II). In fact, at present, designers still need to analyze applications by hand in order to decide which ISEs would be the most beneficial. Although this is still feasible in reasonable time for a single application, when several DSP or cryptographic applications need to be hand accelerated, the existence of an automatic and yet efficient methodology is of major importance. In this paper, a solution is given to this problem, and a set of algorithms is proposed to identify ISEs from automatic application analysis.

For all these reasons, and due to the important progress in design automation, many traditional and new vendors propose customizable processors. Tensilica Xtensa [1], ARC

Manuscript received July 14, 2004; revised March 16, 2005. This paper was recommended by Associate Editor R. Camposano.

L. Pozzi was with the Ecole Polytechnique Fédérate de Lausanne (EPFL), School of Computer and Communication Sciences, Laussane CH-1015, Switzerland. She is now with the Faculty of Informatics, University of Lugano, Lugano 6900, Switzerland (e-mail: Laura.Pozzi@epfl.ch).

K. Atasu was with the Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland. He is now with the Computer Engineering Department, Bogazici University, Istanbul, Turkey (e-mail: atasu@cmpe.boun.edu.tr).

P. Ienne is with the Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne CH-1015, Switzerland (e-mail: Paolo.Ienne@epfl.ch).

Digital Object Identifier 10.1109/TCAD.2005.855950

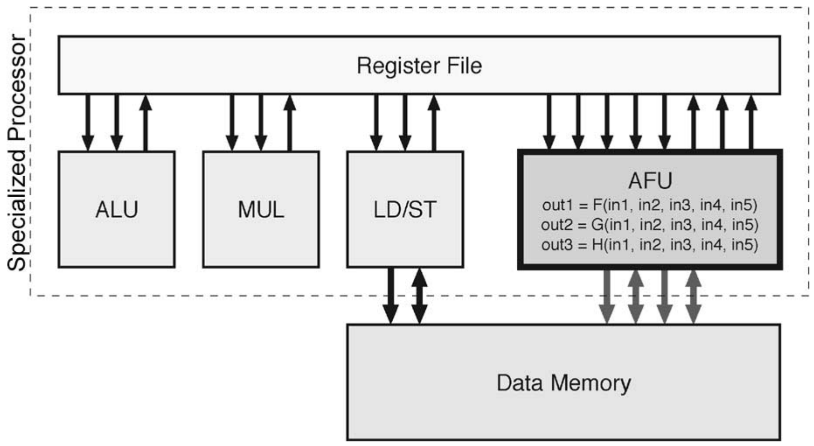

Specialized Processor
Register File
ALU
MUL
LD/ST
AFU
out1 = F(in1, in2, in3, in4, in5)
out2 = G(in1, in2, in3, in4, in5)
out3 = H(in1, in2, in3, in4, in5)
Data Memory

Fig. 1. Typical extensible processor with a five-input three-output application-specific functional unit.

ARCtangent [2], STMicroelectronics ST200 (a.k.a. Lx) [3], MIPS CorExtend [4], and Nios II [5] are only a few of the many application-specific customizable architectures that have emerged in the recent years. These extensible CPUs offer customers the possibility to tailor the processor to a specific application. Roughly, such processor cores accept extensions in the form of application-specific functional units, as shown in Fig. 1. Various processors differ in the number of operands they can supply in a single cycle from the register file, in the number of results they can write back to it, in the availability of memory ports, and in the existence of architecturally visible registers in the additional unit.

In the following section, the state of the art in ISE is described, and the present work is compared to it. The specific goals and contribution of this paper are anticipated in Section III, with the aid of motivational examples aimed at showing the importance of the work. The problem which is here tackled is formalized in Section IV, and Section V introduces the algorithms proposed to solve it. Results are described in the two following sections: Section VI details the experimental setup used, and Section VII discusses the results. This paper concludes with some considerations on future directions opened by this work.

II. Related Work

Loosely stated, the problem of identifying ISEs consists of detecting clusters of operations, which, when implemented as a single complex instruction, maximizes some metric—typically performance. Such clusters must invariably satisfy some constraint; for instance, they must produce a single result or use no more than four input values. The problem solved by the algorithms presented in this paper is formalized in Section IV, but this generic formulation is used here to discuss related work.

In the 1990s, research in design methodologies for system-on-chip processors has been mainly revolving around the synthesis of application-specific instruction-set processors (ASIPs). This includes the automatic generation of complete instruction sets for specific applications [6]–[9]. In that context, the goal is typically to cluster all the atomic operations required for an application into a complete instruction set, which minimizes some important metric (e.g., execution time, program memory size, number of execution units). An example of synthesis of application-specific instructions more closely related to the goals of the present work was discussed for embedded signal processing applications [10]: The purpose 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 favor 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. The philosophy of this paper is different: The goal is here directly formulated as achieving a maximal gain per special instruction. Section III will show the importance of very large clusters in realistic cases.

Some work on reconfigurable processors and, more recently, on customizable architectures is more in line with the goal of this paper. Early attempts used a bottom-up greedy approach to cluster operations together by adding predecessor operations until some constraint is violated [11], [12]. A formal approach has been proven to result into a decomposition of maximal single-output subgraphs [13]; 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. Under limited input–output constraints, Kastrup *et al.* have found useful to disregard maximal size and extract from maximal patterns every possible subpattern—the potential advantage arising from a wider reuse of the special instruction in various parts of the application [14]. An interesting early approach to build all single-sink patterns was used for PipeRench [15]: It consists in iteratively building, with appropriate data structures, all patterns of size l rooted at each node from the patterns of size $l-1$. The complexity of the exponential process was reduced using some heuristics, including limiting the pattern size.

Other authors use approaches combining template generation (identification, in our parlance) and template matching (instruction selection, as it is called in compilers). One such piece of work is peculiar in that clustering is based on the frequency of node type successions—e.g., multiplications followed by additions—rather than on the frequency of execution of specific nodes [16]. This emphasis on global frequency of recurrence has similar effects as in previous work on signal processors [10]: The authors observe that the number of operations per cluster is typically small and conclude that simple pairs of operations appear the best candidates. Successive work by the same group widens the scope of clustering to address parallel templates [17]: The goal is there to cluster not only nodes connected by a dependence edge but also independent nodes that can be scheduled together. Heuristic is used to decide which parallel templates are best clustered. Emphasis is still on relatively small clusters and experiments were limited to five internal nodes. Template generation leading to patterns with arbitrary inputs and outputs was developed for very long instruction word (VLIW) processors [18]. Here too, the number of operations per pattern is restricted in the experiments to a maximum of three, because of the algorithm execution time among other reasons. The importance of growing larger clusters is acknowledged in a more recent work [19]. The authors implement incremental cluster growth but they do not stop the search process as soon as the constraints are violated; to avoid an explosion of the search space, they guide the growth of the cluster with a heuristic function based on operation criticality, operation latency, operator area, and cluster input/output. The use of a heuristic guide function appears effective in reducing the search space, but it makes it hard to ensure that good candidate clusters are not overlooked. Yu and Mitra [20] have goals very similar to those of the present work with one main difference: They focus on connected graphs and thus base their algorithm on the union of cones of the data-flow graphs. Albeit still of exponential complexity, their algorithm solves the simplified problem much faster in practice. We have included a discussion of the benefits of considering disconnected graphs in Section VII.

In the last couple of years, researchers have addressed the variety of configuration possibilities of commercial customizable processors. A recent formulation for the Nios II processor uses an exponential enumeration algorithm to find all patterns with a single output [21]; the algorithm is usable in practice in the given microarchitectural context by limiting the number of inputs to 5. A very comprehensive work, first introduced in 2002, has covered the whole customization flow of the Tensilica Xtensa platform with considerable microarchitectural accuracy [22]. Their template-generation phase includes independent templates, apparently similar to the parallel templates introduced around the same time by Brisk *et al.* [17]. In the iterative process of generating templates, the authors use a prioritized list and, to reduce the number of useless templates generated, they trim the list when the advantage of a template is below some fraction of the advantage of the best one. In another contribution [23], the same authors suggest how to exploit the hierarchical nature of software applications and progressively refine the selected templates. Finally, Tensilica is to our knowledge today the only company to market a tool for the automatic generation of the optimized processor configuration from high-level application code [1]. Among the different customizations supported by the tool, instruction fusion corresponds to what is called here ISE [24]. Exhaustive enumeration of the patterns is used, subject to the usual input–output constraints, as well as to a maximal number of estimated cycles to run the special instruction. The higher potentials of the algorithms presented here, with respect to all above methods, are detailed at the end of Section III.

Baleani *et al.*, in the context of hardware/software partitioning [25], address the identification problem in a manner similar to this paper's. A greedy 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 IV). The exact algorithm presented in this paper is more expensive but considers the complete design space and therefore does not miss potentially good solutions. Finally, note that the run-time performance of the exact algorithm presented here is orders of magnitude better than that of a similar previously reported work [26], because of addition of a pruning criterion based on the number of inputs (see Section V-B2 for details).

III. Motivation and Contributions

The contributions of this paper are motivated by means of two examples. Fig. 2 shows the data-flow graph of the basic block most frequently executed in a typical embedded processor benchmark; nodes represent atomic operations (typically corresponding to standard processor instructions) and edges represent data dependencies. 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. For different reasons, most existing algorithms would bail out before identifying such large subgraphs, and yet, despite the apparently large cluster size, the resulting Functional Unit can be implemented as a very small piece of hardware.

Availability of a further input would include also the following accumulation and saturation operations (subgraph M2 in Fig. 2). 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. Among others, an exact algorithm is presented here that identifies all of the abovementioned instructions depending on the given user constraints. To the best of the knowledge of the authors, the algorithm presented here is the only one described in literature capable of this.

Since the exact algorithm mentioned above is limited in the size of basic blocks it can handle, a second motivational example is presented that shows the importance of heuristic solutions. The core of the aes encryption algorithm is the “round transformation,” operating on a 16-B state and described in Fig. 3. The state is a two-dimensional array of bytes consisting

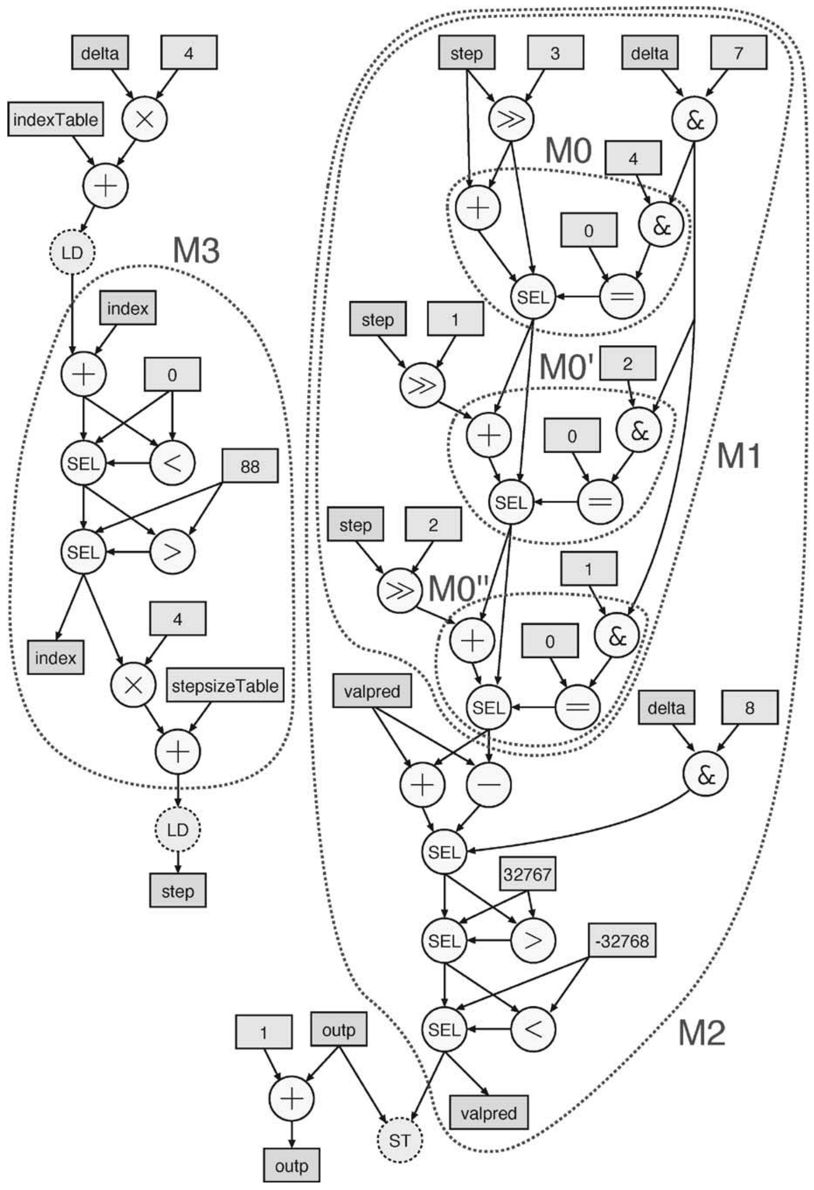


Fig. 2. Motivational example from the adpcmdecode benchmark [27]. SEL represents a selector node and results from applying an if-conversion pass to the code.

of four rows and four columns; the columns are often stored in four 32-bit registers, and are inputs of the round transformation. First, a nonlinear byte substitution is applied on each of the state bytes by making table lookups from substitution tables stored in memory. Next, the rows of the state array are rotated over different offsets, and then a linear transformation called the MixColumn transformation is applied to each column. Finally, an XOR with the round key is performed, and the output of a round transformation becomes the input of the next one. In the C code we have used, the above core has been unrolled twice by the programmer, so that the size of the main basic block amounts to around 700 assemblerlike operations.

The exact algorithms proposed in this paper cannot handle such size for high (e.g., 8–4) I/O constraints. However, applications such as this one are of extreme interest for ISE even for higher I/O constraints: Specialized instructions consisting mostly of very simple bitwise calculations have great potential of improving performance significantly. The MixColumn block (see Fig. 3) represents the most computationally intensive part of the algorithm; it is free of memory access, and is the most interesting candidate for ISE. Moreover, it has very low I/O requirements of 1 input and 1 output (graph M0). Availability of three further inputs would allow inclusion of a RotateRows block (graph M1), and of course all four MixColumns in parallel should be chosen when four input and four output ports are available (graph M2), and all MixColumns and RotateRows when 16 inputs and 4 outputs are given (graph M3).

A heuristic that can automatically identify the abovementioned ISEs is of immediate interest and obvious importance. In this paper, we present several heuristics, the most effective of them based on genetic algorithms, that attempt to perform a choice close to that of expert designers. Section VI will discuss in detail the achievements and limitations of the proposed algorithms with respect to this application.

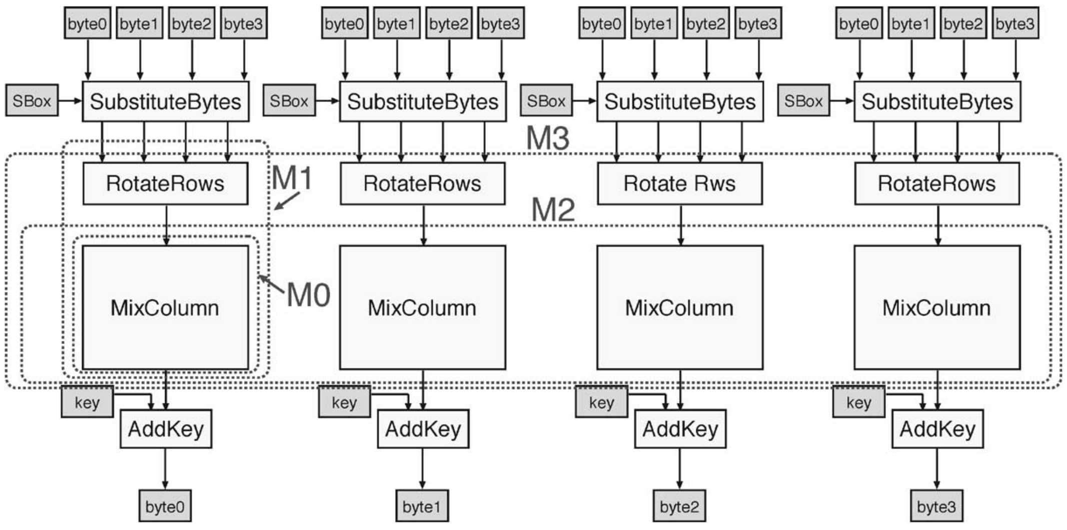


Fig. 3. Motivational example from the aes encryption benchmark.

All contributions of this paper are now finally enumerated. This work improves the state-of-the-art in four respects: Firstly, prior work is mostly limited to instructions with a single output (with the exceptions of a very limited number of outputs and cluster size [17], [25] or of limitation to very specific cases [28]). The techniques presented here identify custom instructions with any number of outputs up to a user-specified constraint. Note that current VLIW architectures like ST200 and TMS320 can commit four values per cycle and per cluster.

Secondly, the present methods can detect any kind of disconnected graphs, which results in the possibility of automatically identifying also single-instruction multiple data (SIMD)-like instructions. Most previous techniques can either only identify connected subgraphs or are limited to small candidates.

Thirdly, heuristics are presented that can identify ISEs consisting of hundreds of nodes and that in some cases closely mimic the choice of expert designers.

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. Two methods are presented, one exact, which is limited in the size of the basic blocks it can process, and some heuristic techniques—the most effective based on genetic algorithms—that can process large numbers of nodes.

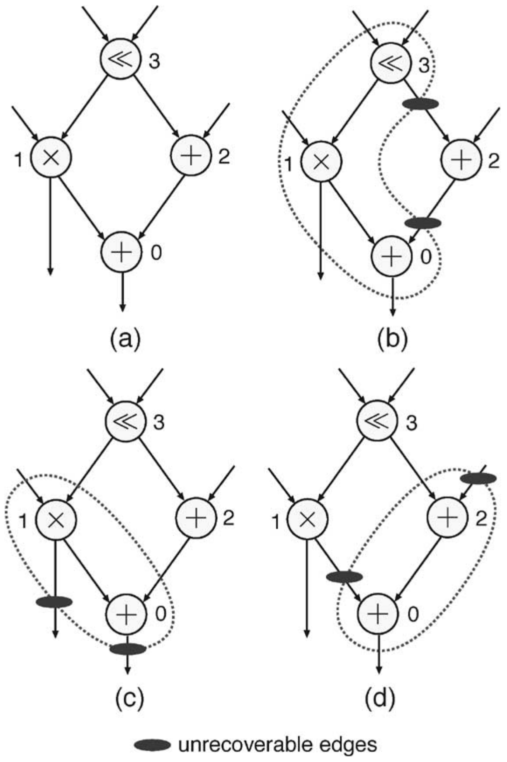


Fig. 4. (a) Topologically sorted graph. (b) Nonconvex cut. (c) A cut violating the output check, for $N_{\text{out}} = 1$. (d) A cut violating the permanent input check, for $N_{\text{in}} = 1$.

IV. Problem Statement

We call $G(V,E)$ the directed acyclic graphs (DAGs) representing the data flow 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 an induced subgraph of G: $S \subseteq G$. There are $2^{|V|}$ possible cuts, where $|V|$ is the number of nodes in G. An arbitrary function $M(S)$ measures the merit of a cut S. It is the objective function of the optimization problem introduced below and typically represents an estimation of the speedup achievable by implementing S as a special instruction.

We call $\text{IN}(S)$ the number of predecessor nodes of those edges that 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, $\text{OUT}(S)$ 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. We call the cut S convex if there exists no path from a node $u \in S$ to another node $v \in S$ that involves a node $w \notin S$. Fig. 4(b) shows an example of a nonconvex cut.

Finally, the values N_{in} and N_{out} express some features of the microarchitecture and indicate the register-file read and write ports, respectively, which can be used by the special instruction. Also, due to microarchitectural constraints, some operation types might not be allowed in a special instruction, depending on the underlying organization chosen. This may reflect, for instance, the existence or absence of memory ports in the functional units: When no memory ports are desirable, load and store nodes must be always excluded from possible cuts, as reflected in the example of Fig. 2 (nodes labeled LD and ST represent memory loads and stores, respectively). We call F (with $F \subseteq V$) the set of forbidden nodes which should never be part of S. We have addressed some particular cases of memory loads in another work [29].

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

Problem 1 (Single-Cut Identification): Given a graph G^+ and the microarchitectural features N_{in}, N_{out}, and F, find the cut S, which maximizes $M(S)$ under the following constraints:

1) $\text{IN}(S) \leq N_{\text{in}}$;
2) $\text{OUT}(S) \leq N_{\text{out}}$;
3) $F \cap S = \emptyset$; and
4) S is convex.

The convexity constraint is a legality check on the cut S and is needed to ensure that a feasible scheduling exists: As Fig. 4(b) 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 dependencies of this graph once S collapsed into a single instruction.

We call N_{bb} the number of basic blocks in an application. Since we will allow several special instructions from all N_{bb} basic blocks, we will need to find up to N_{instr} disjoint cuts, which, together, give the maximum advantage. This problem, referred here as selection, could be 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 as follows.

Problem 2 (Selection): Given the graphs G_i^+ of all basic blocks and the microarchitectural features N_{in}, N_{out}, and F, find up to N_{instr} cuts S_j which maximize $\sum_j M(S)_j$ under the same constraints of problem 1 for each cut S_j.

One should notice that the above formulation implicitly assumes that the benefits of multiple instructions S_j (represented by $M(S)_j$) are perfectly additive. In some architectures, such as RISC and single-issue processors, this is practically exact. In other architectures such as VLIW, some secondary effects might slightly make the merit function nonadditive; yet, the detailed computation of the advantage of several cuts simultaneously would require scheduling the remaining operations and would therefore be infeasible in this context.

The above formulation of the problems addresses ISE exclusively limited to data-flow operations. We attack only indirectly the control flow by applying typical compiler transformations aimed at maximizing the exploitable parallelism, such as if conversion and loop unrolling. We believe that a comprehensive solution of the identification of data-flow clusters is a precondition to the exploration of the control flow. Similarly, the approach presented here does not address the storage of some variables in the functional unit implementing an instruction—in other words, functional units might contain pipeline registers but not architecturally visible ones. The automatic allocation of local storage [29] is a natural extension of the present work.

V. Identification and Selection Algorithms

We propose two kinds of solutions to solve the problem just formalized: Exact methods, described in Section V-A, and approximate methods, described in Section V-B, used when the size of basic blocks prevents the exact methods to complete.

A. Exact Techniques

We introduce the algorithms to solve exactly problems 1 and 2 in three successive steps: 1) find the optimal single cut in a single basic block; 2) find the optimal set of n nonoverlapping cuts in a single basic block; and finally 3) find an optimal set of nonoverlapping cuts in several basic blocks. The first step corresponds to problem 1, the second step to a generalization of problem 1, which is formalized below, and the last step corresponds to the final goal expressed in problem 2.

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. It solves exactly problem 1 above. 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. We write that $u \succ v$. Fig. 4(a) 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 one and zero 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. The nodes of the search tree immediately following a zero branch represent the same cut as their parent node, and can be ignored in the search. Fig. 5 shows the search tree for the example of Fig. 4(a), with some tree nodes labeled 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 toward lower levels; therefore, the search space is pruned.

A trivial case, when it is useless to explore the subtree of a particular node in the search tree, is when such node represents a cut S, which contains a forbidden node in F. Clearly, such S cannot be a solution of problem 1, nor can it be any solution which adds further nodes.

But the real usefulness of this traversal order can be understood from the following discussion. 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. An example of this situation is given

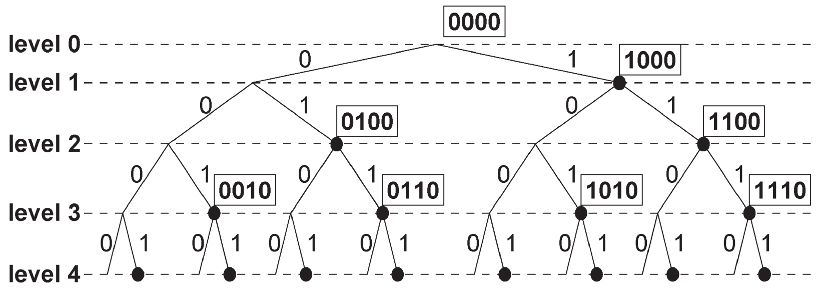


Fig. 5. Search tree corresponding to the graph shown in Fig. 4(a).

in Fig. 4(c), where an output constraint of 1 is violated at inclusion of node 1, and cannot be recovered. 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 Fig. 4(b) 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.

We formalize the above intuition in the following simple theorems, which justify the use of the described traversal order. Given two cuts of S_1 and S_2 of G, we will indicate as $S_1 \cup S_2$ the induced subgraph of G, which contains all nodes in S_1 and in S_2.

Theorem 1 (Monotonicity of the Number of Outputs): Let S_1 and S_2 be two disjunct cuts of G such that for every node $u_1 \in S_1$ and every node $u_2 \in S_2$, it is $u_2 \succ u_1$. Then, $\mathrm{OUT}(S_1 + S_2) \geq \mathrm{OUT}(S_1)$.

Proof: By definition of the number of outputs, there are $\mathrm{OUT}(S_1)$ nodes in S_1 which are predecessors of edges toward nodes of G^+ which are not in S_1. It could be possible that $\mathrm{OUT}(S_1 + S_2) < \mathrm{OUT}(S_1)$, only if some of these predecessor nodes would not be in $S_1 + S_2$ or if their successors would be in $S_1 + S_2$. Neither can happen because all nodes of S_1 are in $S_1 + S_2$ and all nodes in S_2 are predecessors in G of nodes in S_1 (that is, they come later in the ordering). ■

Theorem 2 (Monotonicity of the Convexity): Let S_1 and S_2 be two disjunct cuts of G such that for every node $u_1 \in S_1$ and every node $u_2 \in S_2$, it is $u_1 \prec u_2$. Then, if S_1 is nonconvex, $S_1 + S_2$ is also nonconvex.

Proof: By definition of convexity, if S_1 is nonconvex, there exist at least one node $w \notin S_1$ which is on a path from a node $v \in S_1$ to another node $v \in S_1$. The cut $S_1 + S_2$ can be convex only if at least u and/or v would not be in $S_1 + S_2$ or w would be in $S_1 + S_2$. Neither can happen because all nodes of S_1 are in $S_1 + S_2$, and w is a (indirect) successor of $u \in S_1$, whereas all nodes in S_2 are predecessors in G of all nodes in S_1 (that is, they come later in the ordering). ■

Additionally, we can limit the search space also when specific violations of the input-port constraint happen. For this, we note that there are cases when the edges entering a cut S from G^+ cannot be removed from $\mathrm{IN}(S)$ by further adding

```
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 (forbidden()) return;
    if (!output_port_check()) return;
    if (!permanent_input_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. Single-cut identification algorithm.

to the cut nodes of G which appear later in the topological ordering. We call $\mathrm{IN_f}(S)$ the number of predecessor nodes of those edges which enter S from the rest of G^+ and either: 1) belong to V^+ or F—that is, come from either primary input variables or forbidden nodes or 2) they are nodes that have been already considered in the tree traversal and have been assigned a zero in the cut, i.e., they have been excluded from the cut. In the first case, there is no node in the rest of G that can remove this input value. In the second case, nodes that could remove the inputs exist, but they have been already excluded from the cut. Of course, it is always $\mathrm{IN_f}(S) \leq \mathrm{IN}(S)$ and we call such inputs permanent. As an example, consider Fig. 4(d); after inclusion of node 2, the cut has accumulated two permanent inputs: One is the external input of node 2, and another is the input of node 0, which was made permanent when node 1 was excluded from the cut. Similar to what happens for the output-port and the convexity constraints, when the number of permanent inputs $\mathrm{IN_f}(S)$ violates the input-port constraint when reaching a particular search-tree node, the subtree rooted at that node can be eliminated from the search space.

Fig. 6 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 level of the tree on

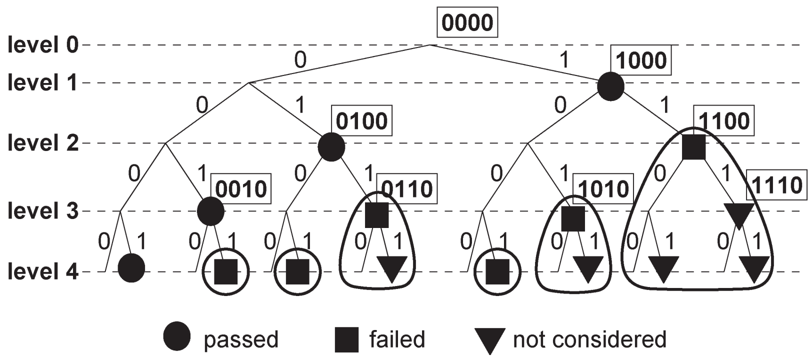


Fig. 7. Execution trace of the single-cut algorithm for the graph given in Fig. 4(a), for $N_{\text{out}} = 1$ and $N_{\text{in}} = 1$. Note that cut 1100 corresponds to the subgraph shown in Fig. 4(c), it violates output constraints, and cut 1010 corresponds to the subgraph shown in Fig. 4(d), it fails the permanent input check.

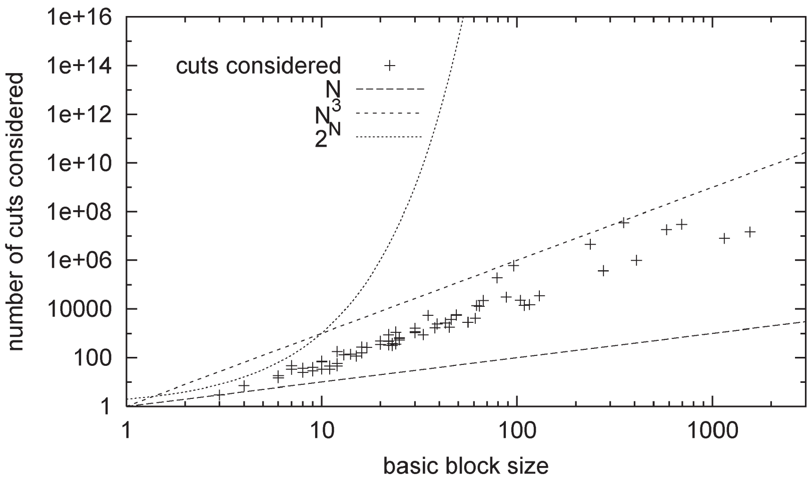


Fig. 8. Number of cuts considered by the single-cut algorithm with $N_{\text{in}} = 4$ and $N_{\text{out}} = 2$, for several graphs (basic blocks) with sizes varying between 2 and 1500 nodes.

which the branch is taken. The function `forbidden` returns true if S contains a node in F. The functions `input_port_check`, `permanent_input_port_check`, and `output_port_check` return a true value when, respectively, $\text{IN}(S) \leq \text{N}_{\text{in}}$, $\text{IN}_{\text{f}}(S) \leq \text{IN}_{\text{in}}$, and $\text{OUT}(S) \leq N_{\text{out}}$. The function `convexity_check` returns a true value when S is convex. When either the output-port check, the permanent input-port check, or the convexity check fail, 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.

Fig. 7 shows the application of the algorithm to the graph given in Fig. 4(a) with $N_{\text{out}} = 1$ and $N_{\text{in}} = 1$. Only four cuts pass the output-port check, the permanent input-port check, and the convexity check, while six cuts are found to violate a constraint, resulting in the elimination of five more cuts. Among 16 possible cuts, only ten are therefore considered (the empty cut is never 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`, `permanent_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 $O(2^{|V|})$ but, although still exponential, the algorithm reduces in practice the search space very tangibly. Fig. 8 shows the run-time performance of the algorithm using an output-port constraint of two and an input-port constraint of four on basic blocks extracted from several benchmarks. The actual performance follows rather a polynomial trend 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.

2) Multiple-Cut Identification: We can easily adapt the algorithm described in the previous section to identify multiple cuts from a single graph. Formally, we generalize Problem 1 as follows:

Problem 3 (Multiple-Cut Identification): Given a graph G^+ and the microarchitectural features N_{in}, N_{out}, and F, find the K disjoint cuts S_j which maximize $\sum_j \text{M}(S_j)$ under the following constraints for each cut S_j:

1) $\text{IN}(S_j) \leq N_{\text{in}}$;
2) $\text{OUT}(S_j) \leq N_{\text{out}}$;
3) $F \cap S_j = \emptyset$; and
4) S_j is convex.

Of course, for $K = 1$, this reduces to problem 1.

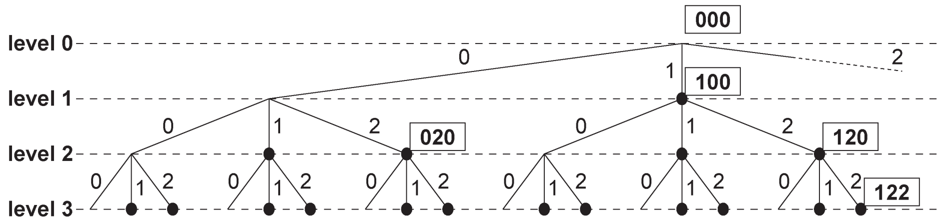


Fig. 9. Search tree for two cuts.

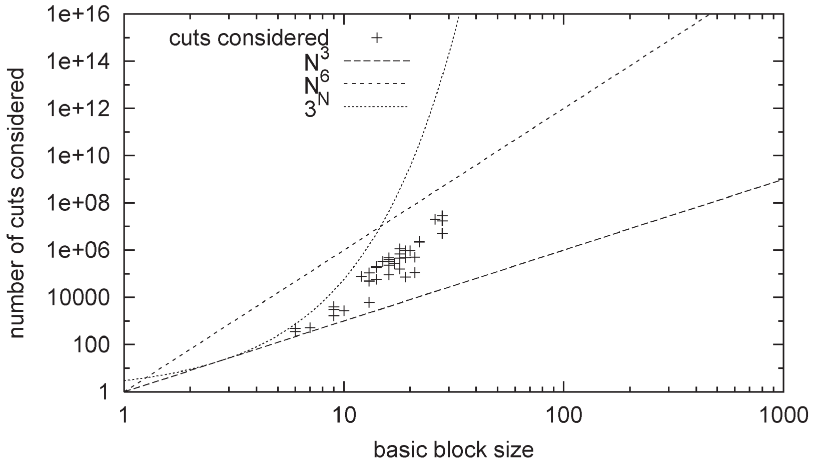


Fig. 10. Number of cuts considered by the multiple-cut identification algorithm with $K = 2$, $N_{\mathrm{in}} = 4$, and $N_{\mathrm{out}} = 2$.

This generalization is easily achieved: If K is the number of cuts to be identified within a basic block, it suffices to build a similar search tree where every node makes $K + 1$ branches instead of 2. Fig. 9 shows a fragment of a tree for $K = 2$. Nodes of the search tree now represent K cuts: An n-branch at level i leads to inclusion of the graph node with index i in the nth cut. Fig. 10 shows the run-time performance of the multiple-cut identification algorithm for $K = 2$, $N_{\mathrm{out}} = 2$, and $N_{\mathrm{in}} = 4$, on basic blocks extracted from several benchmarks. Only basic blocks of up to 30 nodes could be processed within hours.

3) Optimal Selection: The multiple-cut identification algorithm is the natural building block for solving problem 2. One should allocate the N_{instr} cuts to be selected to the various basic blocks. For this, in the general case, one should find out the advantages of allocating 1, 2,... up to N_{instr} to each basic block, and find the best combination. This requires up to $N_{\mathrm{instr}} \cdot N_{\mathrm{bb}}$ applications of the multiple-cut identification algorithm and all possible combinations represent possible solutions.

Choosing the best solution exhaustively after the many identifications is probably feasible in most practical cases, but the main computational burden lays in the number of uses of the expensive multiple-cut identification algorithm. Even considering that in practical cases the maximum merit of most basic blocks is usually below the merit of the good cuts in other basic blocks (property which could reduce the number of basic blocks to be considered to a handful), a fully optimal selection could often be too expensive. We will consider in the next section a slightly approximated selection algorithm, which can be proven optimal in a broad class of cases and which brings the number of invocations to the multiple-cut identification algorithm to at most $N_{\mathrm{bb}} + N_{\mathrm{instr}} - 1$.

B. Approximate Techniques

We begin our series of approximate techniques by addressing the last problem mentioned: The large number of invocations of the multiple-cut identification algorithm required for an optimal selection. We address the problem in Section V-B1 by introducing a modified selection algorithm, which we call pseudooptimal: In fact, it can be proven optimal if some very broad conditions are verified.

Although effective in pruning the search space, this pseudooptimal selection algorithm still relies on the multiple-cut identification algorithm, whose complexity is sensibly higher than that of the single-cut identification (see Fig. 10). In Section V-B2, we therefore introduce a heuristic approach, which reverts back to using the more efficient single-cut identification.

The resulting algorithm is very efficient and effective for basic blocks of medium size—typically up to approximately hundreds of nodes, that is, most of the important basic blocks of typical applications. Nevertheless, especially when one uses aggressive compiler transformations, there are cases where one wishes to address much larger basic blocks. We present two very different techniques to overcome the limitations of the iterative selection algorithm. Firstly, in Section V-B3, we show a heuristic technique to partition large graphs in subgraphs small enough for the iterative selection algorithm to be applicable successfully. Secondly, in Section V-B4, we follow a completely different strategy by employing a genetic algorithm

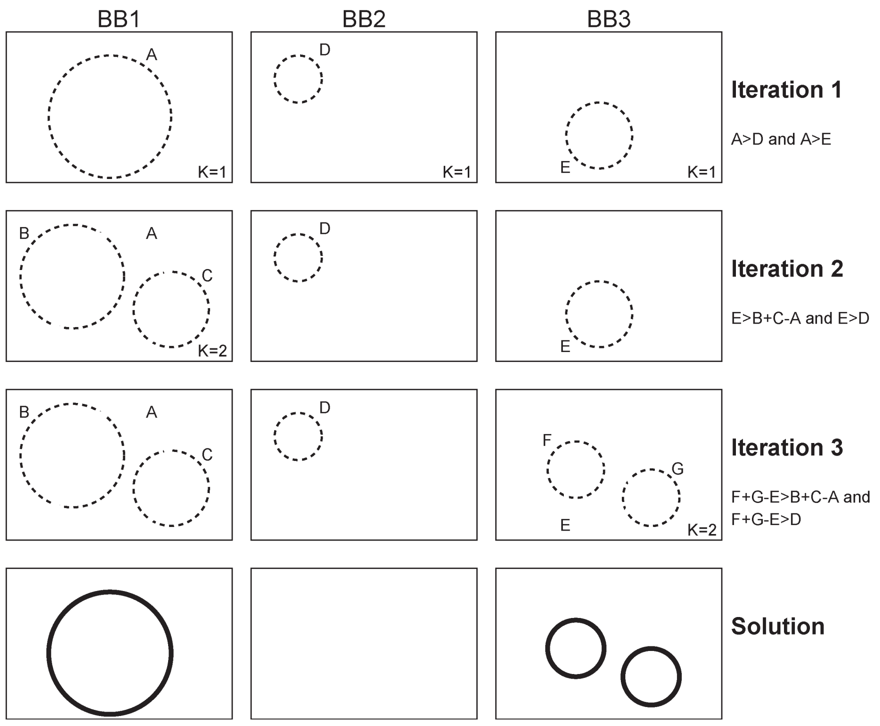


Fig. 11. Pseudooptimal selection of three cuts in three basic blocks. Circles represent cuts, that is, subgraphs, of the basic blocks. Dashed circles are best candidates returned by five calls to the multiple-cut identification algorithm of Section V-B1.

```
pseudo_optimal_selection() {
  for (i = 0; i < BBS; i++) {
    K[i] = 0;
    M0[i] = 0;
    M1[i] = multiple_cuts_ident(i, 1);
  }
  j = index_max_diff(M1[], M0[], BBS);
  K[j]++;
  for (i = 0; i < INSTR - 1; i++) {
    M0[j] = M1[j];
    M1[j] = multiple_cuts_ident(j, K[j] + 1);
    j = index_max_diff(M1[], M0[], BBS);
    K[j]++;
  }
}
```

Fig. 12. Pseudooptimal selection algorithm.

for the selection problem; we will benchmark it, when possible, against the exact solutions.

1) Pseudooptimal Selection: The pseudooptimal selection algorithm begins by applying the single-cut identification algorithm on each basic block ($K = 1$). The first cut is chosen from the basic block, which offers the largest speedup improvement. Then at each iteration, the algorithm: 1) increments the value of K for the basic block, which was chosen by the previous iteration; 2) performs multiple-cut identification on this basic block with the new value of K; and 3) calculates the improvement. Again, the new cut is chosen from the basic block that gives the largest speedup improvement. The iterations continue until N_{instr} cuts are chosen. Fig. 11 illustrates the algorithm with a simple example.

Fig. 12 gives the algorithm in pseudo C notation. The function `multiple_cuts_ident(i, j)` implements the multiple-cut identification algorithm and returns the cumulative benefit $\sum_k \mathrm{M}(S_k)$ of choosing `j` cuts from the `i`th basic block. The function `index_max_diff(A, B, n)` returns the number `i` for which `A[i]` $-$ `B[i]` is maximal, `A` and `B` being two `n`-element vectors. It is clear from Fig. 12 that the algorithm requires application of the multiple-cut identification algorithm at most $N_{\text{bb}} + N_{\text{instr}} - 1$ times.

The algorithm can be proven to return optimal solutions if, for every basic block, the additional advantage of increasing the number of cuts selected in the basic block is a monotonically decreasing function. This can be formalized in the following theorem. For this, let us call G_i^b the incremental merit of adding an ith cut ($i > 0$) to basic block b. Specifically, $G_1^b = M(S)$, where the cut S is returned by solving problem 1, and, for $i \geq 2$, $G_i^b = \sum_j^{\text{best } i \text{ cuts}} \mathrm{M}(S_j^b) - \sum_j^{\text{best } i\, -\, 1 \text{ cuts}} \mathrm{M}(S_j^b)$, where both sums are values returned by solving problem 3 with the multiple-cut identification algorithm on basic block b.

Theorem 3 (Sufficient Condition For Optimality): If, for all basic blocks b, the values G_i^b constitute sequences decreasing with i, then the solution returned by the pseudooptimal algorithm is optimal.

Proof: Selecting the best allocation of the N_{instr} cuts in the N_{bb} basic blocks corresponds in this case to selecting the best N_{instr} additional advantages G_i^b from N_{bb} ordered lists. The pseudooptimal selection algorithm compares at each step the additional advantages possible on all basic blocks and chooses the largest one: The lists being ordered, the cuts are also chosen as if their additional advantage were drawn form a sorted global list of all additional advantages, and therefore their overall merit is maximal (in fact, the pseudooptimal selection algorithm acts like the Merge step of a Merge-Sort algorithm [30]). ■

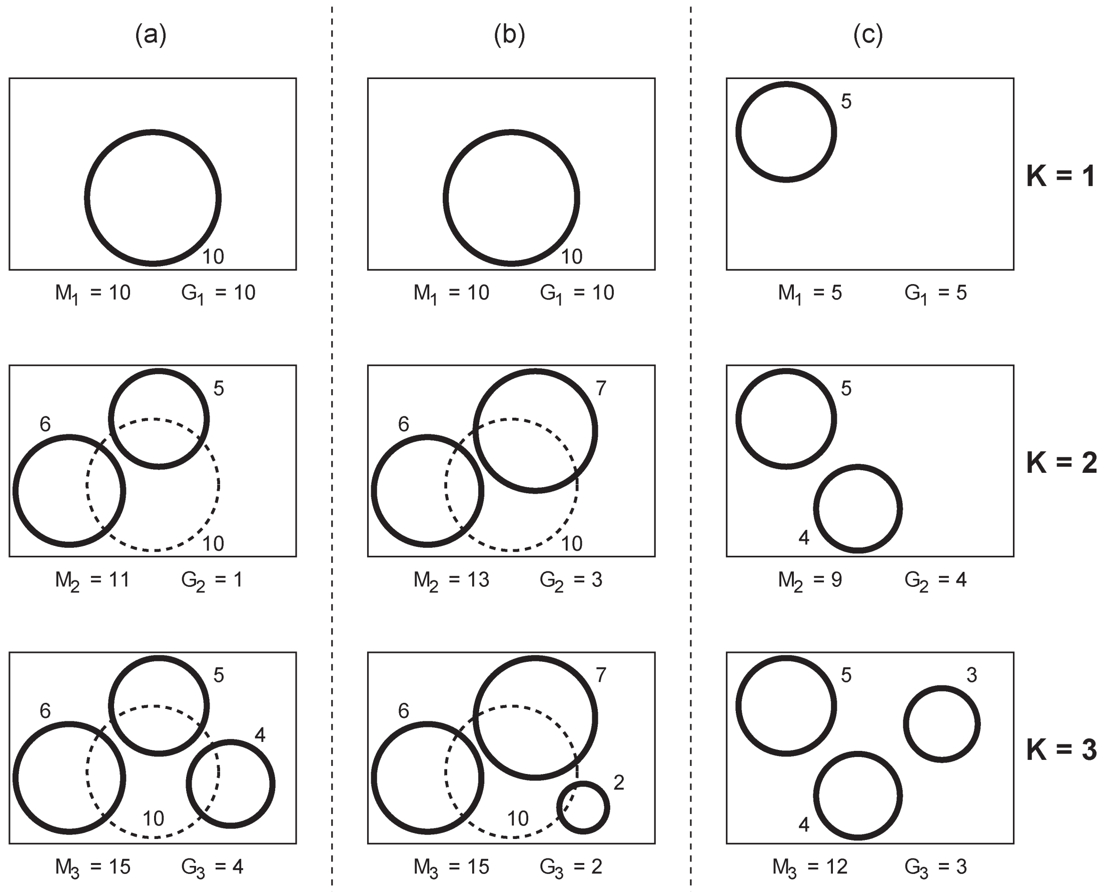


Fig. 13. Three examples of basic blocks with different implications on the optimality of the pseudooptimal selection algorithm. With basic blocks such as (a) pseudooptimal algorithm is not guaranteed to be optimal because the incremental merit of adding a new cut is not monotonically decreasing (i.e., $G_1 = 10$, $G_2 = 1$, and $G_3 = 4$). With basic blocks such as (b) or (c), the selection is optimal because Theorem 3 and, in the latter case, Corollary 1 apply.

It should be noted that the above condition is sufficient but not necessary.

An important case where the above property is verified is expressed by the following corollary.

Corollary 1 (Special Case of Optimality): If, for every basic block, the multiple-cut identification of K cuts results in the identification of the same cuts identified for $K-1$ plus a new cut, the pseudooptimal algorithm returns the optimal solution.

Proof: The hypothesis says that for any basic block b and any K, the multiple-cut identification returns cuts $S_0^b, \ldots, S_{K-1}^b$, when looking for K cuts, and returns the same cuts plus S_K^b, when looking for $K+1$ cuts. Since the identification is optimal, it must be $S_K^b \leq S_j^b$, $\forall j \leq K-1$, or cut S_K^b should have been chosen instead of one of the other cuts when identifying the best K cuts. Therefore, the sequence $G_i^b = S_i^b$ is monotonically decreasing with i and Theorem 3 applies. ∎

Fig. 13 shows graphically three examples of basic blocks. When none of the basic blocks is like those in Fig. 13(a), optimality is guaranteed. Note that we have applied the pseudooptimal algorithm to all the benchmarks shown in the experiment section and to all basic blocks that could be processed: We have observed that the additional advantage of increasing the number of cuts selected in the basic block was always a monotonically decreasing function. The pseudooptimal algorithm was therefore optimal in all our experiments.

2) Iterative Selection: Repeated calls to the multiple-cut identification algorithm on large basic blocks may result in impracticable computational complexity. To avoid this, a heuristic approach was also used; results of the two selection strategies were compared in the experiments and show that the loss is null or negligible in practical cases.

The iterative selection approach consists 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 the forthcoming identification steps. Fig. 14 illustrates the algorithm on the same example used in Fig. 11. One can notice that, contrary to what happens with the pseudooptimal algorithm, once selected, cuts are now permanently added to the solution; of course, the final selection is not necessarily the same.

Fig. 15 gives the algorithm in pseudo C notation. The function `single_cut_ident(i)` implements the single-cut identification algorithm and returns the benefit $M(S)$ of the best cut from the `i`th basic block. The function `collapse_and_forbid_best(i)` transforms the `i`th basic block by merging all nodes of the currently best cut into a single forbidden node. The function `index_max(A,n)` returns the index `i` for which `A[i]` is maximal, `A` being an `n`-element vector. It is clear from Fig. 15 that the algorithm requires application of the single-cut identification algorithm at most $N_{\mathrm{bb}} + N_{\mathrm{instr}} - 1$ times.

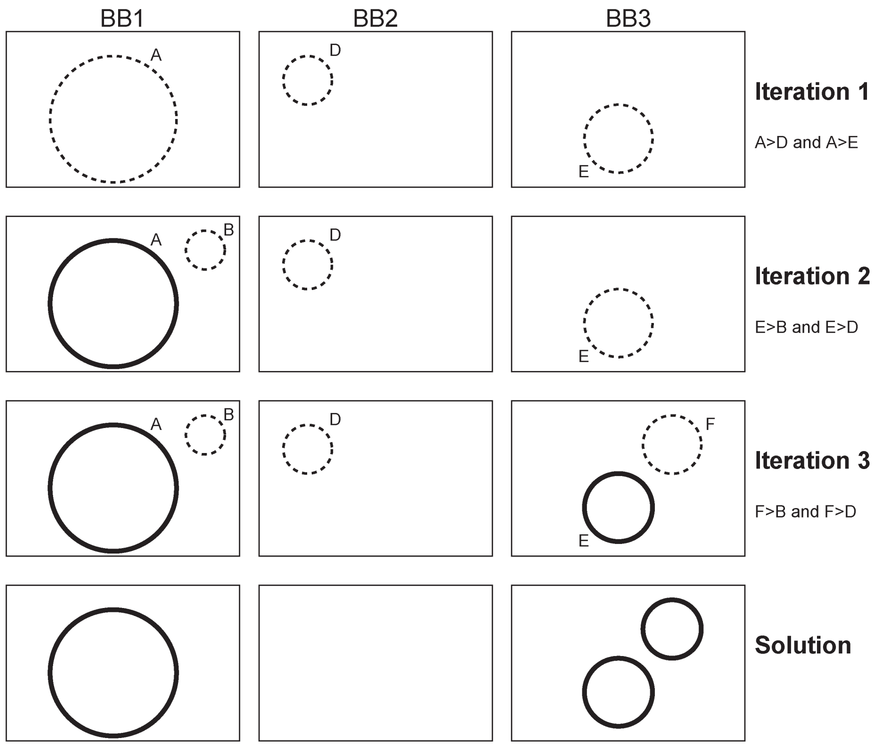


Fig. 14. Selection of three cuts in three basic blocks using the iterative selection algorithm.

```
iterative_selection() {
  for (i = 0; i < BBS; i++) {
    K[i] = 0;
    M[i] = single_cut_ident(i);
  }
  j = index_max(M[], BBS);
  K[j]++;
  for (i = 0; i < INSTR - 1; i++) {
    collapse_and_forbid_best(j);
    M[j] = single_cut_ident(j);
    j = index_max(M[], BBS);
    K[j]++;
  }
}
```

Fig. 15. Iterative selection algorithm.

3) Partitioning-Based Selection: A natural way of extending the optimal single-subgraph-identification algorithm for very large data-flow graphs, where the algorithm performance is not sufficient to explore the complete search space, is to partition the data-flow graphs into pieces that could be handled separately by the algorithm. Intuitively, partitions with the smallest number of inputs and outputs are favorable, since they maximize the chances of locating clusters within the given constraints. Moreover, since the size of the largest partition will determine the overall run-time performance, the size of the partitions should be balanced. Effectively, this is the well-known min-cut hypergraph-partitioning problem.

A hypergraph is a generalization of a graph, where the set of edges is replaced by a set of hyperedges, a hyperedge being any possible set of vertices. We define a hypergraph $H(V^h, E^h)$ based on a data-flow graph $G(V, E)$. The vertex set V^h is identical to the vertex set of G. The hyperedge set E^h is constructed by defining for each node $v \in V$, a hyperedge $h := \{v\} \cup C$, where C is the set of successor nodes of v in G.

We use the recursive bisection algorithm implementation supplied by the hMETIS hypergraph partitioning package [31] to obtain the min-cut partitioning of our hypergraphs. After that, we apply the optimal single-subgraph-identification algorithm of Section V-A1 on each partition. Our strategy heuristically tries to locate the best subgraph within a single partition, and relies on the optimal algorithm for its identification. However, if the best subgraph happens to be split across partitions, there is no way of identifying it, and partial solutions may be strongly suboptimal. Therefore, this heuristic might be perfectly effective in some cases but could completely miss the optimal solution in others. In particular, notice that it can never choose extensions larger than the partition size. To overcome this limitation, we have developed a completely different identification strategy based on genetic algorithms, and we will compare the two approaches in Section VI.

4) Identification With Genetic Algorithms: Introduced in the 1970s by Holland [32], genetic algorithms have emerged as practical and robust optimization methods. In multimodal search landscapes, where several locally optimal solutions exist, genetic algorithms have a high probability of locating the globally optimal solution. Genetic algorithms are stochastic algorithms inspired by the natural phenomena of genetic inheritance and survival of the fittest; they perform a multidirectional search by maintaining a population of potential solutions. In the most common form, solutions are encoded as strings of bits from a binary alphabet. A fitness function evaluates each string, and associates a fitness value with each one, reflecting how good it is. A selection mechanism based on the fitness values decides which individuals will survive to the next generation. Information exchange across different dimensions of the search space is achieved through recombinations based on crossovers

```
genetic_ident_and_selection() {
  P = generate_initial_population();
  evaluate_population(P);
  while (!termination_reached(P)) {
    select_individuals(P);
    apply_genetic_operators(P);
    eliminate_constraint_violations(P);
    replace_individuals(P);
    evaluate_population(P);
  }
}
```

Fig. 16. Genetic algorithm structure.

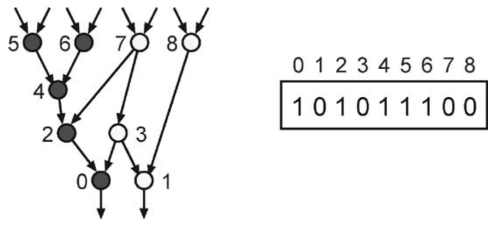


Fig. 17. Genetic algorithm encoding.

and mutations on surviving individuals. Repeated selections and recombinations between generations result in a continuous evolution of the population.

The pseudocode of our genetic algorithm implementation for the single-subgraph-identification problem is given in Fig. 16. In the following sections, we explain its crucial components, and the way they are adapted to our problem.

a) Encoding: A natural encoding of induced subgraphs, within a data-flow graph G, is the use of a binary string of bits with a single bit associated with each graph node. Each node is included in or excluded from the subgraph depending on the value of the corresponding bit. The nodes are assigned an index following a topological order, in the same way as described in Section V-A1: If G contains an edge (u, v) then the index of the bit associated with u is higher than the index of the bit associated with v in the string. Fig. 17 shows an example of encoding.

b) Initial population: The nature of the constraints on the subgraphs makes the generation of an initial population a nontrivial task: Randomly generated bit strings often result in highly infeasible subgraphs. Instead, we generate random clusters of nodes making use of heuristic approaches. Similar to the clubbing algorithm [25], we build convex clusters starting from randomly selected nodes, and add adjacent nodes until the constraints are reached. Another heuristic we employ is the MaxMISO algorithm [13], which extracts all the disjoint maximal-input single-output subgraphs from the data-flow graph. We generate a diversity of initial solutions using the two algorithms.

In terms of quality, the results of the two heuristics mentioned are far from optimal [26], but they have complementary qualities useful to bootstrap our population: The clubbing algorithm can find only small subgraphs. The MaxMISO algorithm, on the other hand, can identify large clusters of operations, not necessarily within the constraints, but possibly containing good legal subclusters, that may appear during future recombinations.

Our strategy is, in fact, similar to the strategy of messy genetic algorithms introduced by Goldberg *et al.* [33]: We explicitly search for low-order, high-fitness solutions as building blocks in the initial stages and then aim to combine such building blocks during recombinations in the later stages.

c) Fitness evaluation: The fitness function is closely related to the objective function, that is, to the function to be optimized—$M(S)$ in our case. However, we also integrate domain-specific knowledge: Application data-flow graphs often contain large clusters starting with a few inputs, generating many intermediate variables, and resulting in a few outputs. In our solutions, we allow subgraphs violating input and output constraints because we expect them to converge eventually to feasible subgraphs, either by growing or shrinking. Growing of subgraphs is implicitly promoted by the characteristics of the objective function; to control this effect, we assign penalties to constraint violations in the fitness function.

The fitness value of a subgraph S satisfying all the constraints is simply equal to the value of the objective function $M(S)$. We penalize all other subgraphs: Let the best fitness value found among the existing feasible solutions be M_{b}. If the number of inputs for the subgraph exceeds the input constraint by $\mathrm{IN_e}(S) = \mathrm{IN}(S) - N_{\mathrm{in}}$ and $\mathrm{OUT_e}(S)$ is the corresponding value for outputs, the fitness value of this subgraph is defined by

$$F(S) = (1 - \alpha\,(\mathrm{IN_e}(S) + \mathrm{OUT_e}(S))) \cdot \min(M_{\mathrm{b}}, M(S)) \quad (1)$$

where α is the penalty parameter. The $\min(\cdot)$ term ensures that the best fitness value is always owned by a feasible solution, and the factor in front of the $\min(\cdot)$ term associates scaled fitness values with infeasible solutions, assigning a penalty increasing with the number of violations.

d) Selection: The selection mechanism models the survival-of-the-fittest phenomenon: The fittest individuals survive and the weaker ones die. Our selection mechanism employs the proportionate selection scheme, using the roulette-wheel selection technique [32]. In the proportionate selection scheme, a solution S with fitness $F(S)$ allocates $F(S)/\overline{F}$ offsprings, where $\overline{F}$ is the average fitness of the population.

e) Genetic operators: The most crucial operation of our genetic algorithm is the crossover operation, that aims at exchanging "building blocks" from different clusters in order to form better subgraphs [see Fig. 18(a)]. Pairs of solutions are chosen randomly from the current population to be subjected to crossover, and the crossover is applied on the pair with a probability called the crossover rate. We found the two-point crossover scheme most suitable in our case. The two-point crossover scheme basically exchanges segments of the two strings between two randomly selected points. It eliminates the single-point crossover bias for the last bits of the strings, and it is more promising in exploiting building blocks of different solutions.

Mutation causes random alterations of the bits of the strings. Each bit is flipped independently with a probability called the mutation rate [see Fig. 18(b)]. Mutation is given a secondary role in our genetic algorithm.

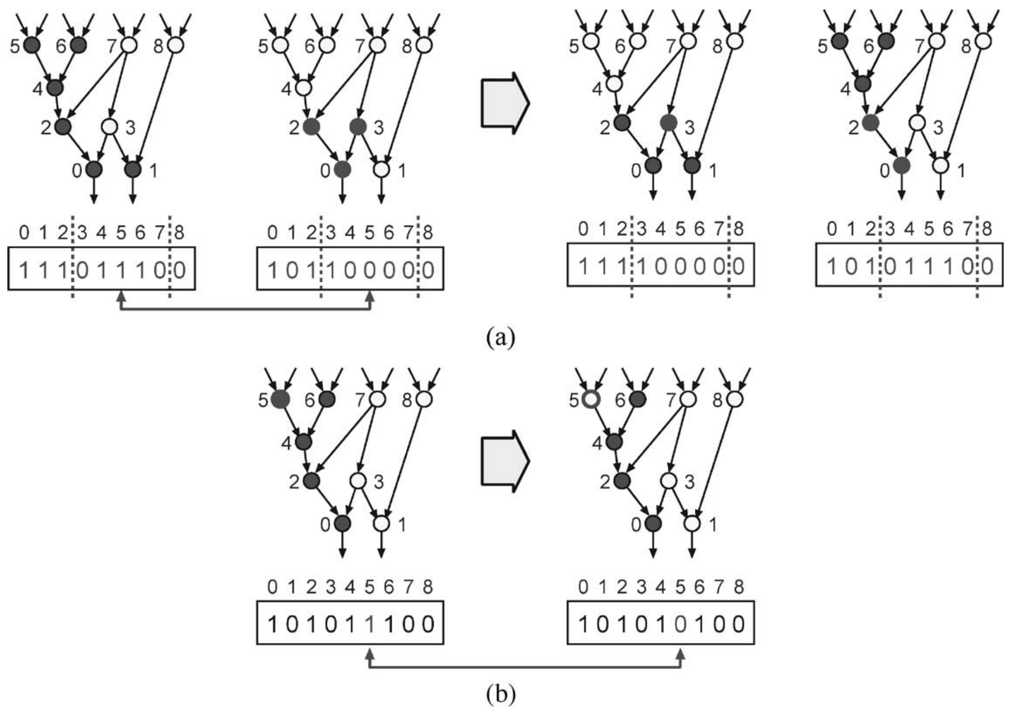


Fig. 18. Genetic operators. (a) Crossover and (b) Mutation.

Operator	Precision	Relative Delay	Relative Area
Multiply-Accumulator	32 bits x 32 bits + 64 bits	1.00	1.00
Adder	4 bits + 4 bits	0.11	0.001
Adder	8 bits + 8 bits	0.12	0.002
Adder	16 bits + 16 bits	0.20	0.003
Adder	24 bits + 24 bits	0.24	0.005
Adder	32 bits + 32 bits	0.25	0.007
Barrel shifter	8 bits	0.08	0.002
Barrel shifter	16 bits	0.11	0.004
Barrel shifter	32 bits	0.16	0.008
Barrel shifter (by constant amount)	any	0.00	0.000
Bitwise multiplexer	any	0.02	0.001

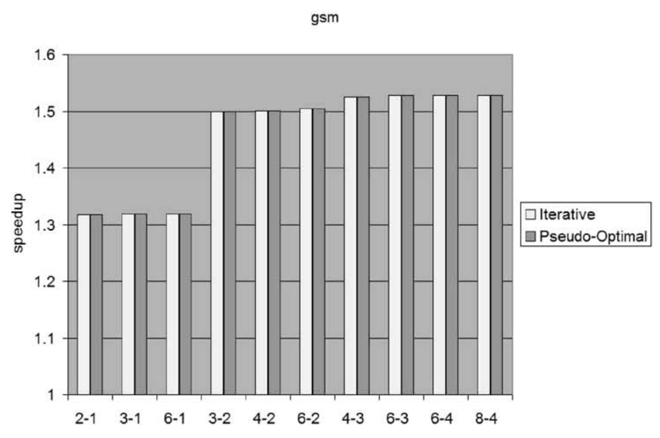
Fig. 19. Examples of hardware timing and area models of some operators. The CMOS technology used is a common 0.18-μm process and the standard cells are from the Artisan library.

f) Constraint violations: The application of genetic operators, on the selected individuals, does not always result in feasible solutions. As mentioned above, input and output violations are taken care by the penalties in the fitness function defined by (1). On the other hand, convexity violations are suppressed immediately. In case a subgraph generated by crossover or mutation is nonconvex, our first attempt is to include all the graph nodes on the violating paths within the subgraph. However, this may not be always possible because of the existence of forbidden nodes, denoted F in Section IV, which should never be part of S (e.g., memory loads). When such a graph node is on a violating path, we remove either all of its predecessors or all of its successors from the subgraph, choosing the solution that removes fewer nodes. This strategy results in convex subgraphs, and is an effective way of combining "building blocks" imported from different solutions.

g) Random immigrants: Genetic algorithms have a natural tendency of converging rapidly. However, when the population converges too quickly into a set of homogeneous solutions, crossovers lose their power in the quest for better individuals; and typically low mutation rates are inadequate for continuing exploration. We make use of the random immigrant mechanism introduced by Grefenstette [34] to compensate this phenomena. The random immigrant mechanism replaces a fraction of the population by randomly generated solutions. The fraction replaced is called the replacement rate. We generate our immigrants in the same way as we generate the initial population. This strategy effectively concentrates mutations in a subpopulation, and plays a role of regenerating the lost genetic material.

Name	# of BBs	size
gsmdecode	15	32
g721encode	9	44
adpcmencode	2	93
adpcmdecode	2	92
viterbi	9	118
fft	9	104
autcor	3	25
bezier	4	35
md5	7	1557
aes	3	696

Fig. 20. Characteristics of the benchmarks used: Number of basic blocks processed and size of the largest basic block.



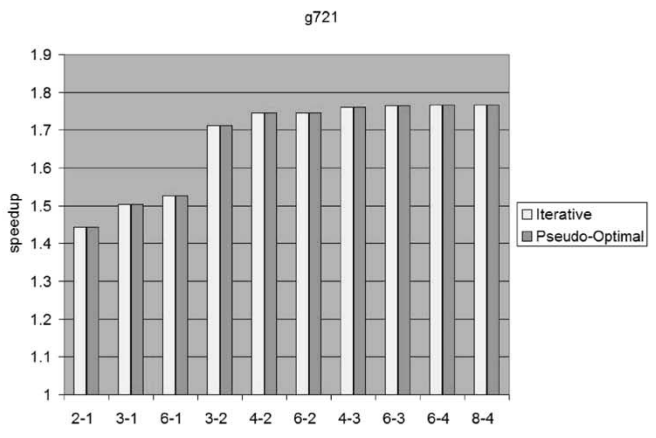


Fig. 21. Comparison of Pseudo–Optimal and Iterative. The difference in performance is negligible, while the difference in complexity is very important (see Figs. 8 and 10).

h) Control parameters: Genetic algorithms, while exploiting the already sampled regions, try to explore new regions of the search space. The balance between the two mechanisms is achieved by a careful selection of control parameters. In our experiments, we observed best results using a rather large population size of 400, a very high crossover rate of 0.95, a modest replacement rate of 0.1, and a low mutation rate of 0.001. The penalty parameter α used in the fitness function is 0.05. Lastly, the termination criteria are defined as having no improvement in the best fitness value for 20 generations.

VI. Experimental Setup

In order to measure the speedup achieved by the algorithms described in this paper, a particular function $M(\cdot)$ is assumed 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 specialized 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 complex complimentary metal–oxide–semiconductor (CMOS) process and normalize to the delay of a 32-bit multiply accumulate. Fig. 19 shows the relative delay and area of some operators.

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 cut critical path. The difference between the software and the 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 the identification algorithms presented here, where the use of a computationally heavier model would be prohibitive.

VII. Results

The described algorithms were implemented within the MachSUIF framework [35] and tested on some MediaBench [27], embedded microprocessor benchmark consortium (EEMBC) [36], and cryptography benchmarks. Application C-code was compiled to MachSUIF intermediate representation, preprocessed with an if-conversion pass, and then analyzed. Fig. 20 describes the benchmarks used and their characteristics: The number of basic blocks selected by profiling, and the number of nodes of the largest basic block.

In order to show the potentials of the algorithms described here, with respect to the state of the art, we have implemented a generalization of Clubbing [25], and the MaxMISO [13] identification algorithms. The first is a greedy linear-complexity algorithm that can detect n-input m-output graphs, where n and m are user given parameters. The second is a linear-complexity algorithm that identifies maximal-size single-output and unbounded-input graphs. This comparison shows the power of the approach presented in this paper when in search for large possibly nonrecurrent ISEs. Note that since the spirit of the presented approach is to find as large as possible ISEs, close to those sought after by experienced designers, recurrence of the identified subgraphs is not checked and is considered to be equal to one by default.

We have also implemented a simple algorithm, denoted with Sequences, that can identify only recurrent sequences of two instructions; it makes possible a comparison with works that propose as ISE's small recurrent subgraphs, such as shift add or multiply accumulate. Sequences traverses the data-flow graph once and selects an ISE every time it identifies a positive-gain two-instruction sequence. Recurrence of the identified sequences is checked, and the gain is modified accordingly, i.e., it is multiplied by recurrence. Of course, the algorithm thus constructed is very simple and less sophisticated than the ones presented in literature, but this simple implementation is meant to give the order of magnitude of the potential gain of such small ISEs, even when recurrence is considered.

Speedup results for all algorithms proposed are shown in Figs. 21, 22, and 27. All graphs depict the speedup of the different algorithms for different benchmarks and for different input and output constraints; note that MaxMISO can be shown for single-output constraints only, and Sequences for three inputs and one output only, while all other algorithms can span any I/O constraint. Partitioning was run only for the cases where Iterative could not complete. In all experiments, selection of up to 16 ISEs was allowed.

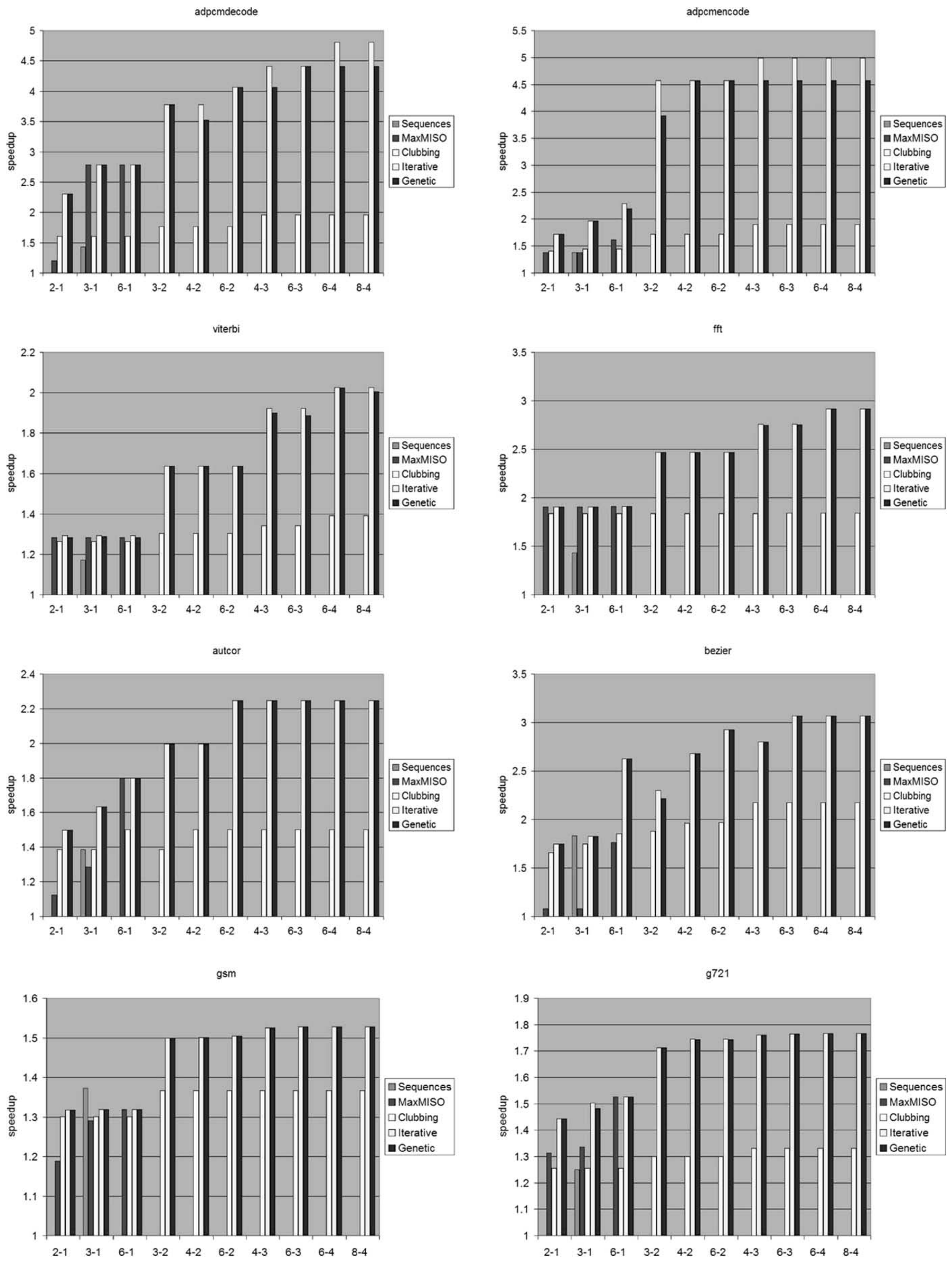


Fig. 22. Analysis of the performance of Iterative.

The conclusions that can be derived by observing the results are organized in three parts: Firstly, a comparison of Pseudo−Optimal and Iterative is shown; recall that Pseudo-Optimal selects multiple ISEs concurrently, and that Iterative identifies iteratively a single ISE at a time (see Section V-B1 and V-B2, respectively for full details). Secondly, the power of Iterative over the state of the art algorithms is demonstrated. Thirdly, the heuristics Genetic and Partitioning (see

	2-1	4-2	4-3	8-4
gsmdecode	0.002	0.001	0.001	0.001
g721encode	0.000	0.003	0.012	0.027
adpcmencode	0.001	0.038	0.837	10
adpcmdecode	0.000	0.025	0.467	4
viterbi	0.001	0.023	0.001	0.003
fft	0.000	0.004	0.027	0.138
autcor	0.000	0.000	0.001	0.001
bezier	0.000	0.004	0.033	0.131
md5	0.200	71	-	-
aes	0.093	77	8865	-

Fig. 23. Run times of Iterative, in seconds, for input/output constraints of 2–1, 4–2, 4–3, and 8–4.

Section V-B4 and V-B3, respectively), are evaluated. Recall that Genetic is a heuristic based on genetic algorithms, while Partitioning is a heuristic that first partitions the original graph and then applies Iterative to each subpart.

A. Comparison of Pseudo-Optimal and Iterative

Fig. 21 depicts two benchmarks, gsmdecode and g721encode, where Pseudo–Optimal was able to terminate. It can be noted that the superiority in performance of Pseudo–Optimal over Iterative is negligible—recall that the observed difference in complexity is instead very important, as shown in Figs. 8 and 10. For all practical purposes, it appears that using Iterative instead of Pseudo–Optimal does not compromise results.

B. Evaluation of Iterative Over the State of the Art

Results for Iterative, Clubbing, MaxMISO, and Sequences can be observed in Fig. 22. Three main points should be noted: Firstly, Iterative constantly outperforms other algorithms; in particular, for low input/output constraints, all algorithms have generally similar performances (see for example benchmarks adpcmencode and viterbi), but in the case of higher, and yet still reasonable, constraints Iterative excels. Secondly, a large performance potential lays in multiple output, and therefore possibly disconnected graphs, and the algorithms presented here are among the first ones to exploit it. Disconnected graphs are indeed chosen by Iterative in most benchmarks; an example is given by adpcmdecode and aes, discussed in detail later. Lastly, note that the potential that lays in the large and non-recurrent graphs chosen by Iterative is constantly much greater than that of small recurrent sequences, especially once the I/O constraints are loosened. This can be seen by observing the performance of Sequences, for example in benchmarks fft and autcor. Recall that current VLIW architectures like ST200 and TMS320 can commit four values per cycle and per cluster: It is therefore reasonable to loosen the constraints.

In the light of the motivation expressed with the help of Fig. 2, it is useful to analyze the case of adpcmdecode.

1) Clubbing is generally limited in the size of the instructions identified, and Sequences selects the couples and–compare and shift–add of subgraphs M0.
2) MaxMISO finds the correct solution (corresponding to M2 in the figure) with a constraint of more than two inputs. Yet, when given only two input ports, it cannot find M1 because M1 is part of the larger three-input MaxMISO M2.
3) Iterative manages to increase the speedup further when multiple outputs are available; in such cases, it chooses a disconnected subgraph at once, consisting M2 + M3. Iterative is the only algorithm that truly adapts to the available microarchitectural constraints.

Of course, the worst case complexity of Iterative is much higher than that of Clubbing, MaxMISO, or Sequences, but it is on average well below exponential complexity, as Fig. 8 shows. The run times of Iterative are shown in Fig. 23, for input/output constraints of 2–1, 4–2, 4–3 and 8–4, and for a single ISE. It can be seen that run times are almost exclusively in the order of seconds or lower; only benchmarks md5 and aes (1500 and 700 nodes, respectively) exhibit run times of the order of minutes for a constraint of 4–2, and of hours or cannot terminate for higher constraints.

Lastly, the benefits of identifying disconnected subgraphs are discussed here. Of course, if only connected subgraphs are considered for ISE identification, the search space decreases considerably. Figs. 24 and 25 show how the run-time complexity decreases in practice when considering connected-only graphs versus possibly disconnected ones. Almost all state-of-the-art proposals consider connected graphs only (see Section II for exceptions), but an algorithm able to identify large disconnected graph, such as Iterative, has the important capability of discovering parallelism. In benchmarks where parallelism is abundant, such as aes, the difference in performance when considering disconnected subgraphs is considerable, as shown in Fig. 26. In particular, note that in the case of connected graphs only, performance fails to increase with the output constraint, i.e., it fails to take advantage of the register port availability.

C. Evaluation of the Heuristics

The performance of the two heuristics proposed, Partitioning and Genetic, is now studied. Note that, for the benchmarks of Fig. 22, Genetic performs very closely and sometimes identically to Iterative. Fig. 27 shows results for the last two benchmarks, md5 and aes, which are the largest and feature 1500- and 700-node basic blocks, respectively. In the case of aes, it can be seen that Genetic underperforms for very small constraints, where Iterative performs well. However, in the cases of higher constraints, Iterative cannot terminate, while Genetic obtains very high speedup, sometimes close to the optimal manual choice, as described in more details later. With respect to Partitioning, it should be noted that Genetic finds systematically better solutions; the difference is extremely significant in the case of the loosest constraints. In aes, for a constraint of eight inputs and four outputs, the best subgraph found by Genetic contains about 130 nodes, whereas Partitioning cannot execute on partitions larger than 80 nodes.

Fig. 28, a detailed version of Fig. 3, depicts the core of the aes benchmark. Iterative finds the optimal solutions of one MixColumn (graph M1) for a constraint of 1–1, a MixColumn and a RotateRows (graph M2) for a constraint of 4–1, three MixColumn transformations in parallel (the equivalent of three

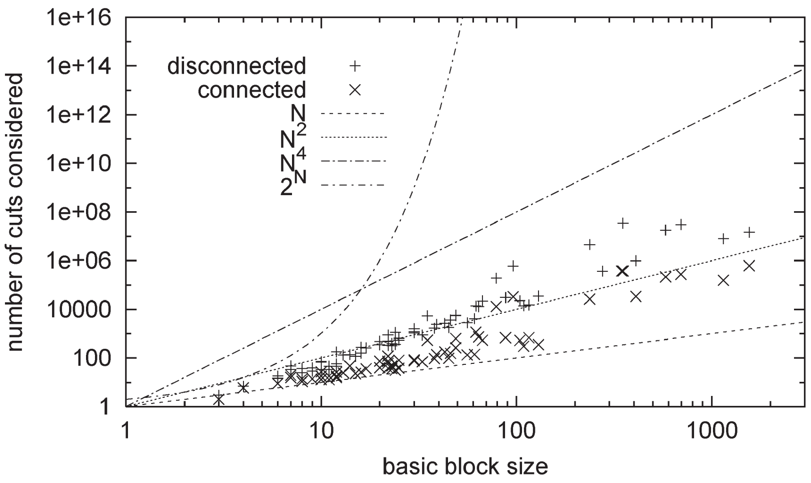


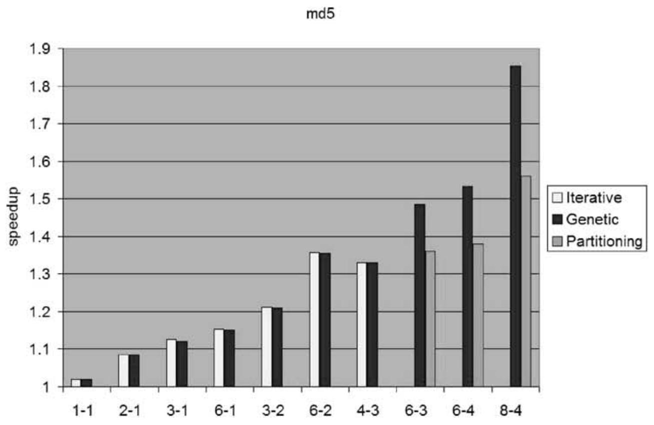
Fig. 24. Number of cuts considered by Iterative when identifying potentially disconnected graph versus connected-only graph, for $N_{\text{in}} = 4$ and $N_{\text{out}} = 2$.

	2-2	4-2	6-2	8-2	2-3	4-3	6-3	8-3	8-4
disconnected	3.8e+6	3.0e+7	1.6e+8	6.4e+8	4.4e+8	2.1e+9	-	-	-
connected	4.0e+4	2.7e+5	8.9e+5	1.9e+6	4.8e+5	3.0e+5	9.6e+5	1.9e+6	1.9e+6

Fig. 25. Complexity difference (in terms of number of cuts considered for identifying the first ISE in the largest basic block of 700 nodes) for potentially disconnected and connected-only graphs, for benchmark aes, and algorithm Iterative.

	2-2	4-2	6-2	8-2	2-3	4-3	6-3	8-3	8-4
disconnected	2.17	2.80	2.93	3.14	2.30	3.14	3.28*	3.41*	3.88*
connected	2.02	2.47	2.76	2.76	2.02	2.47	2.76	2.76	2.76

Fig. 26. Speedup difference for potentially disconnected and connected-only graphs, for benchmark aes, 16 ISEs chosen, and algorithm Iterative ($*$ stands for Genetic, used where Iterative could not terminate).



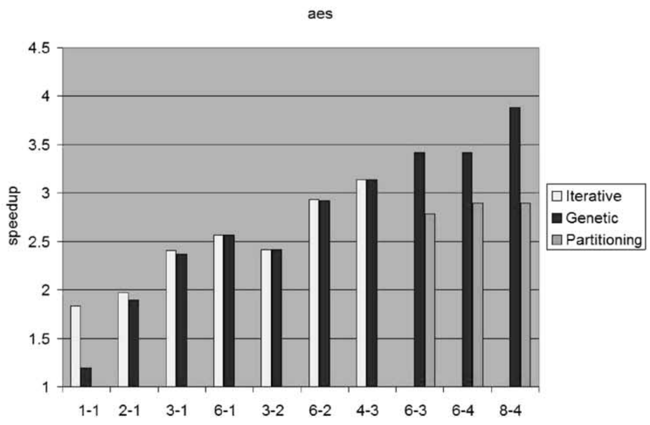


Fig. 27. Evaluation of the heuristic: Genetic and Partitioning. Note that while Genetic underperforms for very small constraints, which can be handled by Iterative, it performs very well, and better than Partitioning for high constraints.

M1 graphs) for a constraint of 3–3, and two RotateRows in parallel (the equivalent of two M2 graphs) for a constraint of 8–2. It cannot terminate for higher constraints. Genetic underperforms for low constraints, but is very effective for higher constraints, intractable for Iterative. It is able to achieve a speedup as high as 3.8x in the case of eight inputs and four outputs, where it identifies three MixColumn transformations in parallel (the equivalent of three M1 graphs), and some bitwise operations from the RotateRows transformations. In terms of run times, Genetic takes seconds for basic blocks of hundreds of nodes, and minutes for aes and md5. The run time does not depend on the I/O constraint.

We now explore in more detail the behavior of Genetic: Fig. 29 demonstrates the maximum, minimum, and average fitness values (normalized to the largest fitness value) obtained applying the algorithm on the kernel of aes, using $N_{\text{in}} = 8$ and $N_{\text{out}} = 4$. It can be seen that the algorithm starts with high-quality initial solutions. MaxMISO identifies separately each MixColumn block together with additional operations coming from the RotateRows transformation, resulting in additional inputs. However, 4 MixColumn blocks within a single ISE, with part of RotateRows, are the desired choice for a constraint of 8–4. The genetic algorithm gradually removes the undesired additional inputs, and tries to combine the MixColumn

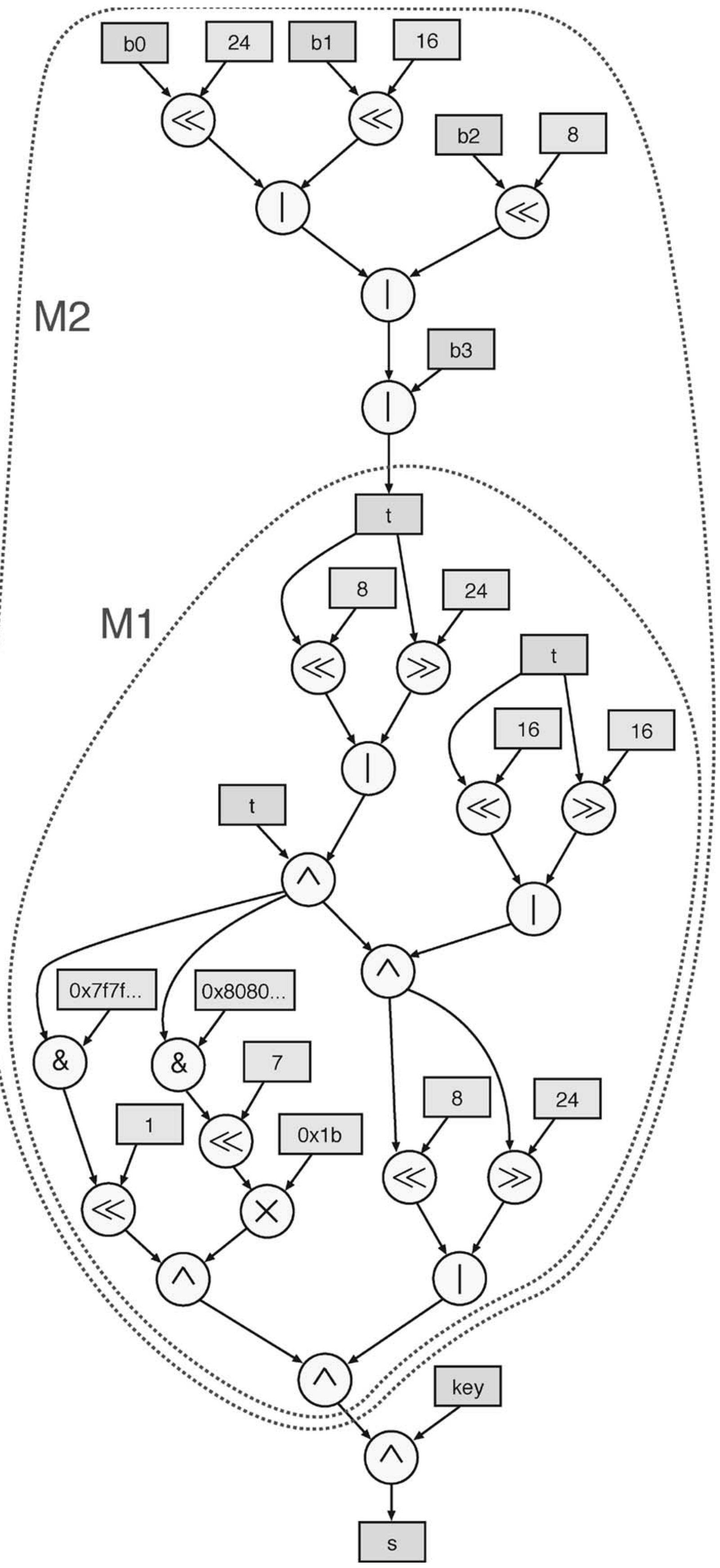


Fig. 28. RotateRows, MixColumn, and AddKey computational blocks of aes (compare with Fig. 3, which shows the same application in less detail). The optimal choice for a constraint of 1–1 is graph M1, which represents a MixColumn transformation, while four MixColumn blocks in parallel should be selected for a constraint of 4–4. MixColumn and RotateRows, graph M2, represent the optimal choice for a constraint of 4–1.

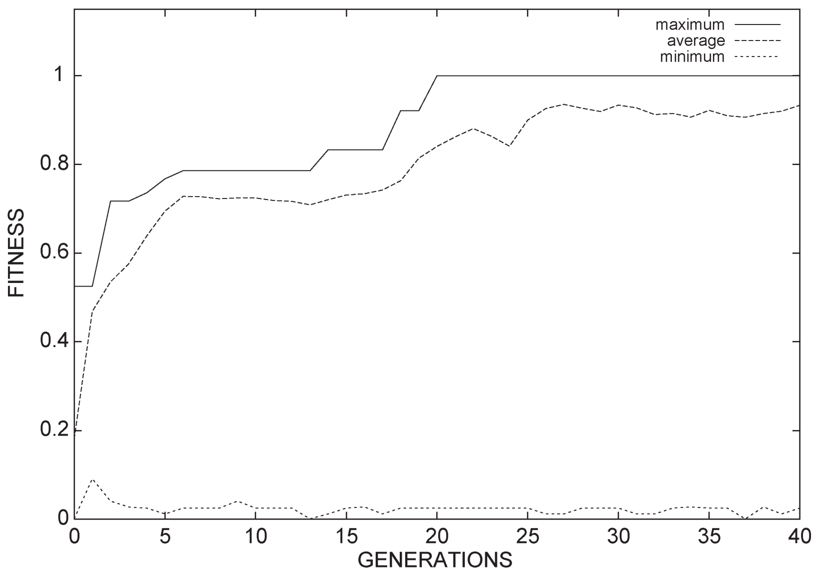


Fig. 29. Maximum, average, and minimum fitness values obtained during identification of the first ISE in aes, using $N_{\mathrm{in}} = 8$ and $N_{\mathrm{out}} = 4$.

blocks. This process is clearly visible in Fig. 29. A second MixColumn block is concatenated with the best initial solution around generation 2, and another one is concatenated around generation 20. The genetic algorithm terminates around generation 40, with the best solution consisting of three MixColumn blocks, and several bitwise operations from RotateRows transformations.

Finally, note that the area investment for implementing 16 ISEs of constraint 4–2, for each benchmark, was within the area of a couple of a 32 $\times$ 32-multiply accumulate.

VIII. Conclusion

This paper has presented new and efficient algorithms for identifying clusters of datal-flow operations to be implemented as application-specific instructions for the existing System-on-Chip processors. This task is essential in order to automate the specialization of commercial processors: Many readily extensible processors are now on the market, and the algorithms presented here automate in an efficient way the identification of their ISEs. In particular, powerful extensions are generated by taking into account microarchitectural constraints given by the designer, and enforcing a legality property on the choice. This work is novel with respect to four points.

1) It considers any register-file read write port constraint; it allows therefore architecture exploration of the best organization with respect to port needs. As a consequence, it is also able to select multiple-output instructions, generally unexplored in the state of the art.
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 formalize identification and selection and solve them together within the same formal framework.
4) It represents a complete methodology in that it provides a powerful heuristic for treatment of large basic blocks, together with exact algorithms for manageable sizes.

The experiments show that the estimated speedup is raised dramatically when compared with the existing state-of-the-art algorithms. The presented exact algorithms efficiently prune the design space, although still exponential in the worst case. To process very large basic blocks, a powerful heuristic solution has been designed around the presented identification algorithm, that performs an ISE selection close to that of manual choice. Future work will address directly the problem of instruction selection under area constraint, inclusion of generic memory elements in ISEs, and definition of the compiler techniques beneficial to ISE generation. Finally, we are planning to use a retargetable compiler to assess precise speedup potentials—especially in VLIW processors where our estimation model is less suitable.

References

[1] T. R. Halfhill, "Tensilicas software makes hardware," *Microprocess. Rep.*, Jun. 23, 2003.

[2] ——, "ARC Cores encourages 'plug-ins'," *Microprocess. Rep.*, Jun. 19, 2000.

[3] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A technology platform for customizable VLIW embedded processing," in *Proc. 27th Annu. Int. Symp. Computer Architecture*, Vancouver, BC, Canada, Jun. 2000, pp. 203–213.

[4] T. R. Halfhill, "MIPS embraces configurable technology," *Microprocess. Rep.*, Mar. 3, 2003.

[5] ——, "Alteras new CPU for FPGAs," *Microprocess. Rep.,* Jun. 28, 2004.

[6] M. Imai, A. Alomary, J. Sato, and N. Hikichi, "An integer programming approach to instruction implementation method selection problem," in *Proc. Eur. Design Automation Conf.*, Hamburg, Germany, Sep. 1992, pp. 106–111.

[7] B. K. Holmer, "Automatic design of computer instruction sets," Ph.D. dissertation, Comput. Sci. Dept., Univ. California, Berkeley, 1993.

[8] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction set definition and instruction selection for ASIPs," in *Proc. 7th Int. Symp. High-Level Synthesis*, Niagara-on-the-Lake, ON, Canada, Apr. 1994, pp. 11–16.

[9] I.-J. Huang and A. M. Despain, "Synthesis of application specific instruction sets," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 6, pp. 663–675, Jun. 1995.

[10] 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 Trans. Comput.*, vol. 48, no. 6, pp. 603–614, Jun. 1999.

[11] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. 27th Int. Symp. Microarchitecture*, San Jose, CA, Nov. 1994, pp. 172–180.

[12] A. Ye, N. Shenoy, and P. Banerjee, "A C compiler for a processor with a reconfigurable functional unit," in *Proc. 8th ACM Int. Symp. Field-Programmable Gate Arrays*, Monterey, CA, Feb. 2000, pp. 95–100.

[13] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG based design approach for reconfigurable VLIW processors," in *Proc. Design, Automation and Test Europe Conf. and Exhibition*, Munich, Germany, Mar. 1999, pp. 778–779.

[14] B. Kastrup, A. Bink, and J. Hoogerbrugge, "ConCISe: A compiler-driven CPLD-based instruction set accelerator," in *Proc. 5th IEEE Symp. Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 1999, p. 92.

[15] S. Cadambi and S. C. Goldstein, "CPR: A configuration profiling tool," in *Proc. 7th IEEE Symp. Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 1999, pp. 104–113.

[16] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 7, no. 4, pp. 605–627, Oct. 2002.

[17] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh, "Instruction generation and regularity extraction for reconfigurable processors," in *Proc. Int. Conf. Compilers, Architectures, and Synthesis Embedded Systems*, Grenoble, France, Oct. 2002, pp. 262–269.

[18] M. Arnold and H. Corporaal, "Designing domain specific processors," in *Proc. 9th Int. Workshop Hardware/Software Codesign*, Copenhagen, Denmark, Apr. 2001, pp. 61–66.

[19] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customisation," in *Proc. 36th Annu. Int. Symp. Microarchitecture*, San Diego, CA, Dec. 2003, pp. 129–140.

[20] P. Yu and T. Mitra, "Scalable custom instructions identification for instructionset extensible processors," in *Proc. Int. Conf. Compilers, Architectures, and Synthesis Embedded Systems*, Washington DC, Sep. 2004, pp. 69–78.

[21] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. ACM/SIGDA 12th Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb. 2004, pp. 183–189.

[22] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 23, no. 2, pp. 216–228, Feb. 2004.

[23] ——, "A scalable application-specific processor synthesis methodology," in *Proc. Int. Conf. Computer Aided Design*, San Jose, CA, Nov. 2003, pp. 283–290.

[24] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proc. Int. Conf. Compilers, Architectures, and Synthesis Embedded Systems*, San Jose, CA, Oct. 2003, pp. 137–147.

[25] 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," in *Proc. 10th Int. Workshop Hardware/Software Codesign*, Estes Park, CO, May 2002, pp. 151–156.

[26] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. 40th Design Automation Conf.*, Anaheim, CA, Jun. 2003, pp. 256–261.

[27] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Annu. Int. Symp. Microarchitecture*, Research Triangle Park, NC, Dec. 1997, pp. 330–335.

[28] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proc. 27th Annu. Int. Symp. Computer Architecture*, Vancouver, BC, Canada, Jun. 2000, pp. 225–235.

[29] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt, "Introduction of local memory elements in instruction set extensions," in *Proc. 41st Design Automation Conf.*, San Diego, CA, Jun. 2004, pp. 729–734.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[31] G. Karypis and V. Kumar, *hMETIS: A Hypergraph Partitioning Package*. Minneapolis: Dept. Comput. Sci. Eng., Univ. Minnesota, Nov. 1998. version 1.5.3.

[32] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: Univ. of Michigan Press, 1975.

[33] D. E. Goldberg, B. Korb, and K. Deb, "Messy genetic algorithms: Motivation, analysis, and first results," *Complex Syst.*, vol. 3, no. 5, pp. 493–530, 1989.

[34] J. J. Grefenstette, "Genetic algorithms for changing environments," in *Proc. Parallel Problem Solving Nature 2*, R. Männer and B. Manderick, Eds. Amsterdam, The Netherlands, 1992, pp. 137–144.

[35] M. D. Smith, and G. Holloway. (2000). *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization.* Cambridge, MA: Harvard Univ., http://www.eecs.harvard.edu/hube/software/

[36] T. R. Halfhill, "EEMBC releases first benchmarks," *Microprocess. Rep.*, May 1, 2000.

Laura Pozzi (M'01) received the M.S. and Ph.D. degrees in computer engineering from Politecnico di Milano, Milan, Italy, in 1996 and 2000, respectively.

She is an Assistant Professor at the Faculty of Informatics, University of Lugano, Lugano, Switzerland. Previously, she was a Postdoctoral Researcher at the School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland, a Research Engineer with STMicroelectronics, San Jose, CA, and an Industrial Visitor at the University of California-Berkeley. Her research interests include automating embedded processor customization, high-performance compiler techniques, and reconfigurable computing.

Dr. Pozzi was the recipient of the Best Paper Award in the embedded systems category at the 2003 Design Automation Conference (DAC). She is in the Technical Program Committee of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES).

Kubilay Atasu (S'03) received the B.Sc. degree in computer engineering from Bogazici University, Turkey, in 2000, and the M.Eng. degree in embedded systems design from Advanced Learning and Research Institute (ALaRI), University of Lugano, Switzerland, in 2002. He is currently working toward the Ph.D. degree at the Department of Computer Engineering, Bogazici University, Istanbul, Turkey.

His research interests include computer arithmetic, cryptography, and hardware/software codesign. He was with the Swiss Federal Institute of Technology Lausanne (EPFL), Processor Architecture Laboratory, in 2003.

Mr. Atasu was a recipient of a Best Paper Award in embedded systems category at the 2003 Design Automation Conference (DAC). He is a Scholar of Turkish Scientific and Technical Research Council (TUBITAK) as of 2004.

Paolo Ienne (S'90–M'96) received the Dottore in Ingegneria Elettronica degree from Politecnico di Milano, Milan, Italy, in 1991, and the Ph.D. degree from the Swiss Federal Institute of Technology Lausanne (EPFL), Lausanne, Switzerland, in 1996.

From 1990 to 1991, he was a Junior Researcher with Brunel University, Uxbridge, U.K. From 1992 to 1996, he was a Research Assistant at the Microcomputing Laboratory (LAMI) and at the MANTRA Center for Neuro-Mimetic Systems of the EPFL. In December 1996, he joined the Semiconductors Group of Siemens AG, Munich, Germany (which is now Infineon Technologies AG). After working on datapath generation tools, he became Head of the embedded memory unit in the Design Libraries division. Since 2000, he has been a Professor at the EPFL and heads the Processor Architecture Laboratory (LAP). His research interests include various aspects of computer and processor architecture, reconfigurable computing, language-based very large scale integration (VLSI) design methodologies, and computer arithmetic.

Dr. Ienne was a recipient of the 40th Design Automation Conference Best Paper Award in 2003. He is or has also been a member of the program committees of international workshops and conferences, including Design Automation and Test in Europe (DATE), the International Conference on Computer Aided Design (ICCAD), and the Workshop on Application-Specific Processors (WASP).