Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors

Paolo Bonzini, *Student Member, IEEE,* Laura Pozzi, *Member, IEEE*

***Abstract*—Automatic generation of a customized instruction set, starting from an input application code, is a complex problem that has received considerable attention in the past few years. Because of its complexity, only simplified versions of the problem have been solved exactly so far. For example, exact algorithms have been proposed for custom instruction identification, but that do not consider recurrence; or methods exist that can indeed handle recurrence, but are limited in how complex an instruction they can identify. However, an exact solution that can handle identification and recurrence simultaneously has been missing.**

We divide the problem into several parts, and concentrate on *covering*, that is selecting a set of non-overlapping and possibly recurrent custom instructions to be implemented and used. We then propose a range of novel algorithms, both exact and approximate, to solve the covering problem in conjunction with the recurrence of candidate extensions.

We propose an optimal search technique that uses branch-and-bound to improve an existing solution, in conjunction with a greedy search to help the algorithm out of any local optima, and achieve a tangible improvement over non-recurrency-aware covering.

***Index Terms*—Application-specific microprocessors, customizable microprocessors, instruction selection, instruction-set extensions, toolchain automation**

I. Introduction

WHILE a variety of designs for customizable accelerators have been proposed by researchers, the most common design of a configurable processor is still the instruction-set extensible processor. In such as system, it is possible to augment the instruction set with a number of extensions that are programmed with an HDL and synthesized into a reconfigurable fabric or into an ASIC.

Tools can aid the design of accelerators, but they cannot yet fully automate it. This is because the fundamental problem of *automatically generating the best performing instruction set extension* is difficult. It requires the tool to enumerate candidate instruction set extensions, understanding how to map them to hardware and what the cost of a hardware implementation is, and finally selecting which extensions will be used and where.

A full solution to the problem should take into account two conflicting factors: the gain of an instruction, and the recurrence of the instruction. Larger extensions achieve higher gains but can be used less often; smaller extensions (while yielding a smaller benefit for each occurrence) can be found many times in the same or even in different applications. Therefore, balancing the two is crucial to obtaining optimal solutions. In addition, complex data-flow topologies often make the best solution unobvious.

In this paper, we formalize the problem of recurrence-aware instruction set extension, and present a framework to partition the solution into phases. This enables usage of existing algorithms for candidate enumeration and isomorphism testing, in order to concentrate on *covering*, that is selecting the non-overlapping custom instructions that are to be implemented and used.

We propose three covering algorithms: a greedy solution, an exact algorithm, and a heuristic with intermediate complexity. Our evaluation shows that the latter achieves improved speedups compared to the greedy solution, while having a very limited time cost when compared to the often prohibitive exact solution.

The remainder of the paper is organized as follows. Section II surveys related work on this topic. Section III presents a compilation flow for instruction set extensible processors, and formalizes the particular problem that we solve in this paper. Section IV details the algorithms we propose, which are validated in sections V and VI. Section VII finally concludes the paper.

II. Related work

Many different variables affect the complexity of finding effective instruction set extensions (ISEs): for example, the size of the candidates (number of outputs or operations), the flexibility of the cost metric, or whether we consider recurrence in the evaluation of a candidate's gain. Past literature in general considered only a subset of these factors, leading in general to solutions that are only valid for simplified instances of the problem.

For example, [1] proposes an exact algorithm to find the best-performing instruction set extension in a basic block, but uses a greedy methodology when finding more than one instruction and does not take recurrency into account.

[2] also did not consider recurrency in the evaluation of a candidate; however, since some treatment of recurrences is necessary in order to achieve significant speedups, it suggested ways to retrofit isomorphism tests in a non-recurrence-aware search algorithm. As in [3], such tests are performed during subgraph enumeration; the result is different from what we propose here in the following ways. First of all, since enumeration is inherently done one basic block at a time, it is not

Manuscript received October 26, 2007, revised March 18, 2008.

Paolo Bonzini and Laura Pozzi are with the Faculty of Informatics, University of Lugano, Lugano 6900, Switzerland (e-mail: paolo.bonzini@lu.unisi.ch, laura.pozzi@unisi.ch).

Digital Object Identifier.

possible to detect isomorphic extensions appearing in different basic blocks. Secondly, these works would pick the single most profitable ISE independent of the number of occurrences. In other words, they prefer large extensions appearing only once, to smaller ones where the number of replicas guarantees a higher gain. Our main result is a set of techniques that are able to find large *and* recurring instruction set extensions.

Other literature achieves better reuse of the instructions or covering of the basic block, but restricts the search to simpler custom instructions. In particular, considering only single-output candidates [4] reduces the problem complexity substantially.

Kastner [5] presents an algorithm that can simultaneously generate instruction set extensions and find their recurrences. Their approach supports multiple-output extensions, but since the aim of their technique is to maximize the number of covered nodes, they cannot plug in an arbitrary measure of an extension's gain. Also, in order to keep the problem tractable, they perform graph covering using a locally optimal heuristic.

Clark [6] also partitions the solution into several subproblems and presents an optimal branch-and-bound covering algorithm. This algorithm uses subgraph size as its merit function, and relies on sorting the candidates in decreasing order of size; the algorithms we present similarly sort the candidates in decreasing order of merit. On one hand the bound computation presented in [6] is more effective than what we present; on the other hand, it does not generalize to other merit functions. The importance of being able to plug in an arbitrary merit function is discussed in Section III-C.

Our work builds on the compilation flows proposed in these works as well as in [7]. All of these, in particular, partition the solution into several subproblems, thus enabling the usage of well-known optimal algorithms for subgraph enumeration [8] or graph isomorphism [9].

Of course, this work does not mark in any way the "end of the road". [10] for example pointed out a possible improvement to instruction-set extension that has not been investigated more deeply. There, symbolic algebra is used to rewrite the application according to the generated instruction set, thus enabling wider reuse of the extensions.

Also, unlike [4], we do not consider instruction overlapping, i.e. duplicating the same operation across multiple ISEs, and only generate extended instructions covering a disjoint set of nodes. While this provides the opportunity to achieve higher speedups, it can also limit the gain in power consumption that customizable processor can achieve.

III. Overview of the Compilation Flow

Our approach partitions the problem solution into 4 phases: subgraph enumeration, isomorphism detection, technology mapping and graph covering, as shown in Figure 1. The problem is solved optimally if exact algorithms are employed in all cases. However, heuristics can be used in any of the phases to reduce complexity.

The rest of this section explains the steps in the compilation flow.

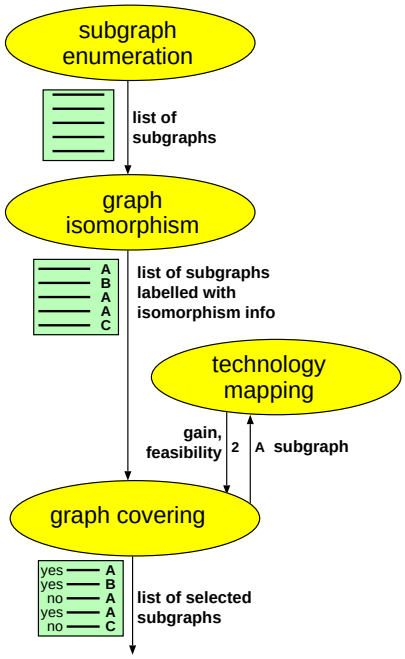


Fig. 1. Proposed compilation flow for recurrence-aware instruction set extension.

A. Subgraph enumeration

The first phase, subgraph enumeration, reads an application intermediate representation and generates the set of all potential custom instructions (subgraphs). Subgraph enumeration is mostly a solved problem; it is possible to enumerate all the potential subgraphs in a few seconds, for basic blocks of hundreds of nodes [8], [11], [12].

While feasible in general, full enumeration is not the only possible choice. Less general techniques [13] or even greedy algorithms [14] may still be a good match for some kinds of accelerator.

B. Detecting isomorphic subgraphs

The second phase reads a list of all potential subgraphs and detects isomorphism among those subgraphs. Isomorphism detection is an important step in most targets as it enables hardware reuse; however, most previous works either do not consider it, or do so in a very limited scope.

As for subgraph enumeration, we do not specify a definition of isomorphism. Most past work on the subject looked for structurally isomorphic subgraphs, either using generic graph isomorphism algorithms (*nauty*, *vf2*) or crafting them specially for data-flow graphs. In principle, however, the compilation flow can also accommodate tests for behavioral equivalence. For example, graph rewriting (*subgraph subsuming* in [2]) or algebraic [10] techniques could be applied to some or all of the subgraphs in the list, and structural isomorphism could be verified on the new list. However, this is a complex and open research problem for which we do not attempt a solution here.

C. Graph covering

The graph covering pass completes the solution by finally selecting a set of non-overlapping custom instructions to

actually be implemented and used. The final set is chosen based on the output of the previous phases as well as the candidates' gains, which are computed by a technology mapping subroutine.

Covering received wide attention in past literature. For simple accelerators that only accept tree-shaped candidates, it is possible to use classic instruction selection algorithms based on dynamic programming [15]. For more complex accelerators, most known algorithms are greedy in one way or another, e.g. [2].

One of the inputs of the covering problem is a *merit function* (see Problem 1). The merit value is usually a measure of speedup, such as the number of cycles of software execution of a candidate, minus the critical path length of the candidate's hardware implementation.

Optimal branch-and-bound covering algorithms can be used if a candidate's merit value is simply given by the number of nodes in the candidate [6]. This special case is important because it achieves optimality if the search is limited to one-cycle instructions, and the cost of executing any node in software is also the unit cost[1] In general, however, different factors can complicate the merit function:

- the cost of hardware execution can be higher than one cycle, especially if synchronous components (such as memories) are included in the ISE, or if the register file bandwidth is a bottleneck [16], [17];
- the cost of software execution can be fractional and thus not proportional to the basic block size. For example, when if-conversion is used, the cost of operations in the conditionally-executed basic blocks should be scaled by the basic blocks' execution frequencies;
- the cost of software operations included in ISEs might be higher than one cycle, for example when custom instruction can include load nodes. [18] This is the case in this paper, since in our setup the accelerator may include read-only memories.

The remainder of this paper will solve a specific formulation of covering, whose formalization and solution will be presented in the following section.

IV. Methodology

The problem solved in this paper is formalized in the following. It can be better understood with the aid of Figure 2, which shows an instance of the problem.

Formally, the inputs to covering are:

- a set of graphs, $G = \{G_1, G_2, ..., G_i\}$, representing the application's basic blocks.
- the execution frequency f_i for each graph G_i, to be gathered by profiling,
- sets of subgraphs $S_i = \{S_{i1}, S_{i2}, ..., S_{ij}\}$ representing all potential instruction set extension instances within each basic block G_i. We will call S the set of all subgraphs S_i in the application, taken from all basic blocks: $S = \bigcup_{G_i \in G} S_i$.

[1]In this case the gain of a candidate of size s is $s - 1$. Adding 1 to the gain of each and every candidate does not change the choices of the covering algorithm, so the subgraph size s itself can be used as the merit function.

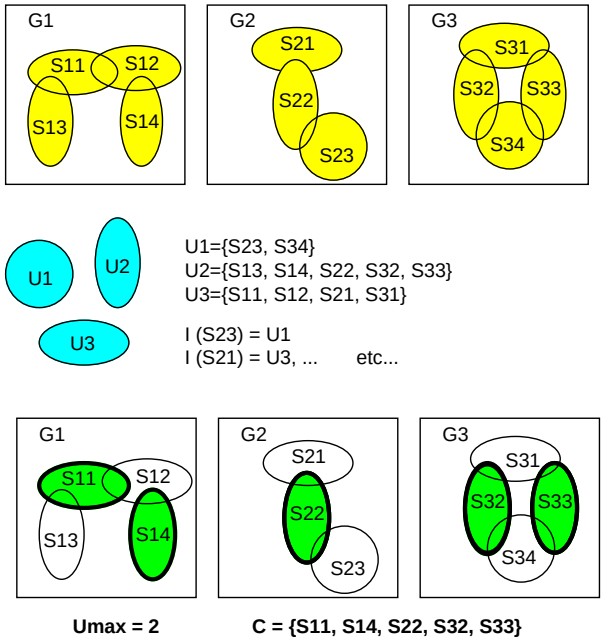


Fig. 2. Understanding the problem formalization. Within an application consisting of basic blocks G_1, G_2 and G_3, the set of all feasible subgraphs S_{ij} (potential instruction set extensions instances) have been enumerated. They have been partitioned into three equivalence classes corresponding to unique patterns U_1, U_2, and U_3, according to isomorphism. A valid solution to the problem, for $U_{max} = 2$, is that of selecting the set C of custom instruction instances highlighted in figure.

- isomorphism information, consisting of a partition of S into n equivalence classes $U_1, U_2, ..., U_n$ (all of the elements of an equivalence class being isomorphic to each other); the equivalence class of S_{ij} is written $I(S_{ij})$.
- a merit function $M(\cdot)$ returning the gain of a subgraph S_{ij}, for example the number of cycles saved by the subgraph when executed as a custom instruction. If two graphs are isomorphic (they are in the same class), the merit function has the same value.

The problem to be solved can be stated as follows.

Problem 1 (Bounded recurrence-aware graph covering): *Select a subset C of S (representing the chosen custom instruction occurrences) such that the overall merit $\sum_{S_{ij} \in C} f_i M(S_{ij})$ is maximized, and the following constraints are respected:*

- *the chosen subgraphs do not overlap:*

$$S_{ij} \cap S_{ik} = \emptyset \ \forall S_{ij}, S_{ik} \in C$$

- *the chosen instruction instances correspond to at most U_{max} unique patterns:*

$$|\{U_k : \exists S_{ij} \in C,\ I(S_{ij}) = U_k\}| \leq U_{max}$$

In the remainder of this section we will describe our contribution to the solution of Problem 1. We first describe a greedy solution, and then derive a practical optimal algorithm, as well as how to use it as part of a faster heuristic.

A. Greedy solution

Our first contribution is a greedy technique that reworks the Iterative algorithm of [8] in order to deal with isomorphic

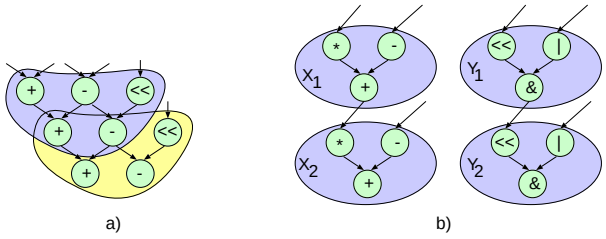


Fig. 3. Overlapping isomorphic subgraphs.

subgraphs. As in the original algorithm, covering proceeds greedily, selecting at every step a locally optimal candidate ISE (a set of non-overlapping isomorphic subgraphs). Nevertheless, even this simple solution has to deal with the entire added complexity of recurrency-aware covering, as this section will explain.

Two phases alternate during the execution of our algorithm.

Finding a valid covering for a candidate. This phase determines which occurrences of a candidate shall be chosen, in case some replicas of a single candidate overlap each other. When selecting multiple ISEs, this phase also discards subgraphs that overlap a previously chosen extension.

Overlapping occurrences of the same candidate have two basic causes, shown in Figure 3. Firstly, as in Figure 3(a), they can happen because the computation has several rounds (e.g. in an unrolled loop), and the end of one round overlaps with the beginning of the next. In this case, if the dark area is chosen as part of an ISE, the light part will not—even though the two areas include isomorphic subgraphs.

Another common case occurs when the subgraph includes multiple disconnected components. The computation in Figure 3(b), for example, produces four candidates after enumeration, corresponding to $X_1 \cup Y_1$, $X_2 \cup Y_1$, $X_1 \cup Y_2$, $X_2 \cup Y_2$. As in the previous case, whenever a program node appears in multiple isomorphic subgraphs, and the algorithm should only pick one of them: in the case of Figure 3(b) only two replicas can be chosen without generating overlap. There is an additional problem: if the second and third were chosen, the two replicas would "feed each other", generating a cycle in the data flow graph. This is not allowed, and these cases have to be ruled out.

As shown in Figure 7 (procedure COVER-SINGLE-CANDIDATE), a single loop can eliminate replicas that would either overlap other occurrences, or generate cycles. Subgraphs are sorted according to the outputs, and non-overlapping subgraphs are picked greedily. The sorting phase allows the greedy algorithm to choose the best combination in cases like Figure 3(b).

Determining if another candidate could possibly yield a higher gain. For a greedy algorithm, all that is left is to recognize whether or not the output of COVER-SINGLE-CANDIDATE does return the single best possible ISE. If this is not the case, the algorithm shall find a valid covering for another group of isomorphic subgraphs. The remainder of this section will show how it is possible to limit the number of calls to COVER-SINGLE-CANDIDATE.

We call a candidate *valid* if we already computed its covering; otherwise the candidate is termed *invalid*. The overall gain of a candidate is the sum of all the gains from each occurrence; and since COVER-SINGLE-CANDIDATE will only discard some copies of the subgraph, it may only decrease the gain of the candidate. Therefore, it is still possible and useful to compute the gain of an invalid candidate—keeping in mind that it may overestimate the real figure.

Our algorithm repeatedly looks at the candidate with the highest maximum gain. If valid, it can be chosen immediately; otherwise the gain may be overestimated, and we need to find the candidate's cover using COVER-SINGLE-CANDIDATE.

An efficient implementation of the algorithm can use a heap data structure to choose the highest-gain candidate. The algorithm using this data structure is also shown in Figure 7.

B. Exhaustive search

An optimal solution to Problem 1 could be achieved by exhaustive search on the whole universe of subgraphs S. Given $|S|$ candidate ISEs, the worst-case complexity of this approach is $O\left(|S|^{U_{max}}\right)$. Of course, since $|S|$ can be as high as 10^6, this has to be augmented with pruning strategies such as branch-and-bound.

We developed a branch-and-bound criterion for this algorithm, described in Figure 4(a). It relies on running the search after the candidates have been grouped into equivalence classes. The equivalence classes $U_1, ..., U_n$, furthermore, are sorted according to the maximum gain that they can contribute (from highest to lowest), and higher-gain classes are always tried first.

Let $g_1, ..., g_n$ be the gain of each equivalence class. Then, at any point, if C is a partial solution, g is its gain, and the next equivalence class to be examined is U_k, the gain bound is $g_{max} = g + \sum_{k \le i < k + U_{max} - |C|} g_k$. If the best gain achieved so far is greater than or equal to g_{max}, it is useless to examine U_k or any equivalence class that follows it.

Once a leaf of the search tree is reached, it is necessary to evaluate the gain of the entire solution; as in the greedy solution, this checks for the presence of overlapping copies and removes them if necessary. In order to solve this subproblem exactly, a *conflict graph* can be built among the copies, and a maximum independent set (MIS) of the graph can be solved (Figure 5)[2].

A secondary pruning criterion, shown in Figure 4(b), can be used to avoid evaluating the gain of each and every leaf of the search tree. To this end, for each subgraph in the solution (including overlapping copies) we compute a *per-node gain*:

$$m_{ij} = \begin{cases} f_i M(S_{ij})/|S_{ij}| & \text{if } I(S_{ij}) \text{ is part of the solution} \\ 0 & \text{otherwise} \end{cases}$$

Then, we compute the maximum gain for each node n, which is $\max_{i,j:n \in S_{ij}} m_{ij}$, and sum these values over all the nodes in the application. In other words, we look at subgraphs that could cover that particular node, and take the highest

[2]While we did implement an exact MIS algorithm [19], we did not use it in our experiments because the faster greedy algorithm (COVER-SINGLE-CANDIDATE in Figure 7) always found the optimal solution for our benchmarks. Of course, this may not be true in general.

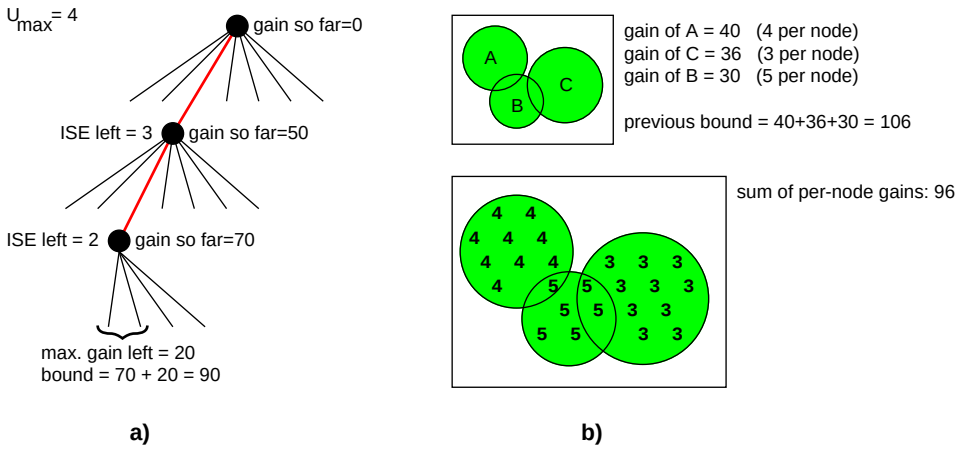
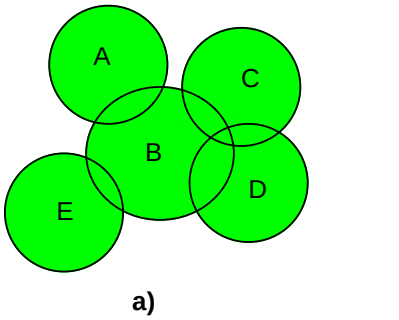


Fig. 4. Pruned exhaustive search. a) ISEs are examined in order of decreasing size. When n search levels are left, the bound is given by the current gain, plus the gain of the next n ISEs to be examined. b) For search tree leaves, the bound is refined to skip the evaluation of MIS for most of the leaves. A covers 10 nodes, B covers 6, C covers 12 (total 28), but together they only cover 25 nodes; the "sum of per-node gains" criterion only counts the three overlapped nodes once.



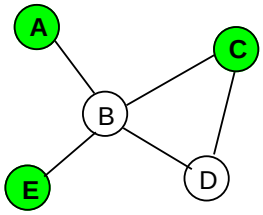


Fig. 5. Evaluating the gain of a search tree leaf. a) A set of overlapping custom instructions. b) In the corresponding conflict graph an edge connects two overlapping occurrences; an optimal choice of extensions corresponds to a maximum independent set of the conflict graph.

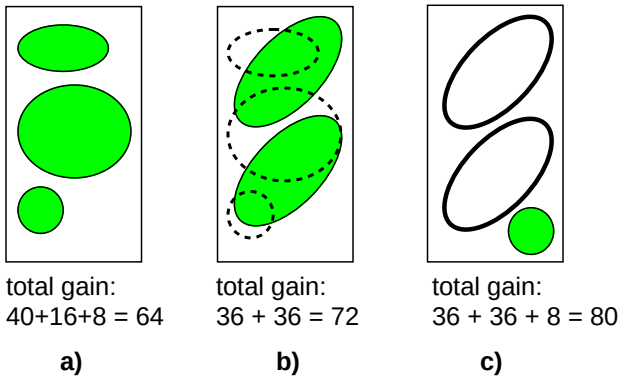


Fig. 6. Sample execution of the optimal hybrid algorithm. (a) Result of greedy search. (b) Exact search is run on a reduced universe to improve the previous solution. (c) Greedy search is run again to find solutions not explored by the previous steps.

per-node gain. The gains of all the nodes are then summed, yielding an upper bound to the solution's gain[3].

Unlike the previous criterion, this one is only useful for search tree leaves and does not allow to prune the search; however, it provides a better estimate of the solution's gain, and therefore it decreases the number of MIS instances to be solved.

C. Optimal hybrid search

Despite these improvements, however, the previous technique is still impractically expensive. For all but the smallest inputs, even a solution with $U_{max} = 2$ may take too much time. For this reason, we propose to use exhaustive search as a subroutine in a different algorithm. This one is more practical, and couples exhaustive search (on a reduced universe) with greedy search; the former is responsible of improving a pre-existing solution, the latter avoids that the search gets stuck on a local optimum.

This algorithm is based on the following observation:

[3]Rather than redoing work unnecessarily for every leaf, it is of course possible to compute the gains and their sum incrementally during the exploration of the search tree.

THEOREM 1: *There is a node that is covered both by the greedy solution and by an optimal solution.*

Proof: Let C be a greedy solution to problem, where $\{UC_1, ...UC_{U_{max}}\}$ are the candidates in C, grouped according to their equivalence class, and $g_1 \geq g_2 \geq \ldots \geq g_{U_{max}}$ are the total gains achieved by each equivalence class. If the greedy solution is optimal, the theorem is trivially true.

Otherwise, we can use the following proof by contradiction. Let C^+ be an optimal solution that is disjoint from C.

Let UC_k^+ be the highest-gain set of isomorphic subgraphs in C^+, and let its gain be g_k^+; k is the lowest value such that $g_k^+ > g_i \ \forall i : k \leq i \leq U_{max}$. Such a set exists because the total gain of C^+ is higher than the total gain of C. Since UC_k^+ has no node in common with any subgraph in C, a greedy algorithm should have considered and chosen UC_k^+ instead of UC_k. ■

This theorem is the basis of the algorithm in Figure 8. The algorithm applies the exact branch-and-bound algorithm to a restricted universe consisting *only of the candidates overlapping a pre-existing solution*; this universe often consists of a few hundred or less candidates, making exhaustive search fast enough to be practical. If the exact algorithm finds a

better solution, it is followed by another greedy run to include candidates that do *not* overlap the currently chosen ones—i.e., candidates that were not considered—and so on until the solution is not improved.

An example execution for $U_{max} = 3$ is found in Figure 6. Figure 6(a) shows the solution S_1 found by the greedy algorithm, consisting of three ISEs that, together, achieve a gain of 64. Figure 6(b) shows how exhaustive search examined candidates overlapping the current solution, and found a new covering S_2 that has only two ISEs and still achieves a better gain. A further greedy run adds a third ISE to the current covering, and creates solution S_3(Figure 6(c)). The algorithm then iterates; if exhaustive search cannot find a better solution, S_3 is optimal and is returned.

We proceed to prove the liveness and correctness of the algorithm.

THEOREM 2: *The algorithm terminates.*

Proof: Both the exact and the greedy runs will return a pre-existing covering if they cannot find a better one (for example, they won't oscillate between two equivalent solutions): therefore, each run can only improve the gain or reduce the number of distinct ISEs (which cannot happen U_{max} or more times). Since the maximum gain achievable is bounded, the algorithm will converge. ■

THEOREM 3: *The algorithm computes an optimal solution.*

Proof: Let us consider the properties of C after line 6; let $\{UC_1, ...UC_{U_{max}}\}$ be the candidates in C, grouped according to their equivalence class, and $g_1, ..., g_{U_{max}}$ be the total gains achieved by each equivalence class.

If an optimal solution covers only nodes that C covers, C is also optimal. There cannot be a better solution with a subgraph covering some nodes in C and some outside C: the exact search at line 6 would have included that subgraph and found the optimal solution.

There could exist a better cover C^+, including a set of isomorphic subgraphs that is disjoint from all the subgraphs in C. We call this set UC_k^+ and its gain g_k^+; k is the lowest value such that $g_k^+ > g_i \ \forall i : k \leq i \leq U_{max}$. Such a set exists because the total gain of C^+ is higher than the total gain of C.

Since UC_k^+ has no node in common with any subgraph in C, UC_k^+ (or another ISE which is also disjoint from C and has higher gain) it will be chosen before UC_k by the greedy run at line 8. In either case the algorithm, while not finding the optimal solution, will recognize this and run another cycle. ■

D. Approximate (d-optimal) hybrid search

The algorithm in the previous section is a substantial improvement over naïve branch-and-bound, as it allows to search more instruction set extensions (usually between 3 and 6) in a matter of minutes. However, its scalability is also limited.

Therefore, we propose using a hybrid algorithm, whose pseudocode is in Figure 9. In this technique, optimal search is run to select a number of candidates $d \leq U_{max}$; d is called the depth of the search. However, if $d < U_{max}$, *only one* candidate (the best one) is selected, and search is started again on the remaining parts of the input. In other words, candidates are chosen one at a time greedily, but "with an eye" to the next few choices.

This trick keeps most of the efficiency of the greedy algorithm, because depths as small as 2 or 3 are in general enough to outperform greedy search, but is also able to dodge its disadvantages. In the remainder of this paper we'll refer to the hybrid algorithm with depth d as *d-optimal*, since it can obviously find an optimal result if $U_{max} = d$.

V. EXPERIMENTAL SCENARIO

In order to evaluate the proposed techniques, we implemented them in a toolchain for extensible processors based on GCC and SimpleScalar/ARM. Our extension of SimpleScalar can accept the definition of up to seven ISEs, each with up to four inputs and up to two outputs; dually, the compiler is able to generate the new instructions.

In the compiler, the ISE identification flow follows the scheme outlined in section III and splits the optimization pipeline in two. Before ISE identification, each operation in the source code is mapped one-to-one with the compiler's intermediate representation. ISE identification, then, has two purposes. First, it produces a description for the custom instructions that can be dynamically linked into the simulator. Second, the compiler can then collapse into a single node the computations that are part of a single custom instruction, so that they are treated atomically in the rest of the compilation.

The first two phases of the compilation flow, namely enumeration and isomorphism check, employ exact algorithms. For enumeration we adapted the search algorithm from [8] in order to output all candidate subgraphs. For isomorphism we used *vf2* [9], a bottom-up algorithm that is well suited to graphs that exhibit regularity, and has a good memory bound (linear in the size of the graph).

The implementation of the proposed algorithms require fast manipulation of the sets U_i from the problem input. Since these are combination sets on the nodes of the data-flow graphs, we used zero-suppressed binary decision diagrams [20]. These allow a memory efficient representation of combination sets and support complex operations on them, including extraction of overlapping sets from a universe. [21]

For comparison with the state of the art, we also implemented a non-recurrency-aware approach, as explained in the related work section. This approach repeatedly picks the highest-gain candidate in a basic block, until it has found U_{max} distinct extensions. Therefore, it does not attempt to solve recurrence-aware identification.

VI. RESULTS

We present results for a series of benchmarks taken from the MiBench [22] suites. We tuned SimpleScalar to match the architecture of the XScale, a single-issue ARM implementation with in-order execution only. The same datasets were used for the profiling run and for the optimized run.

```
COVER-SINGLE-CANDIDATE(candidate, chosen)
    ▷ candidate is a set isomorphic subgraphs
    ▷ chosen is a set of nodes that were previously
          chosen as part of an ISE
    N = chosen
    O = {}
    OCC = {}
    for each element S in candidate do
        if N ∩ S = ∅  ∧
           there are no o1, o2 ∈ O, s1, s2 ∈ S such that
           there is a path from o1 to s1
           and a path from s2 to o2  then
              N = N ∪ S
              O = O ∪ outputs of S
              OCC = OCC ∪ {S}
    return OCC
```

```
COVER(candidates, chosen)
    ▷ candidates is a set of sets. Each element of
          candidates includes many isomorphic subgraphs
    ▷ chosen is a set of nodes that were previously
          chosen as part of an ISE
    H = HEAP-NEW
    V = {}
    for each element c in candidates  do
        HEAP-INSERT-NODE(H, Gain(c), c)
    while HEAP-MAX(H) ∉ V  do
        c = HEAP-EXTRACT-MAX(H)
        OCC = COVER-SINGLE-CANDIDATE(c, chosen)
        HEAP-INSERT-NODE(H, Gain(OCC), OCC)
        V = V ∪ {OCC}
    return HEAP-MAX(H)
```

Fig. 7. A greedy covering algorithm.

```
ITERATIVE-OPTIMAL(S, count)
1  C_greedy = GREEDY(S, count)
2  repeat
3        C = C_greedy
4        repeat
5              C_prev = C
6              C = BRANCH-BOUND-OPTIMAL({s ∈ S : ∃s' ∈ C_prev,  s ∩ s' ≠ ∅}, count)
7        while C ≠ C_prev
8         C_greedy = GREEDY({s ∈ S : ∀s' ∈ C,  s ∩ s' = ∅} ∪ C, count)
9  while C_greedy better than C
```

Fig. 8. An iterative, optimal covering algorithm

```
HYBRID(S, count, depth)
 1  C = ∅
 2  for i = 0 to max(0, count − depth)  do
 3        n = min(count − i, depth)
 4        C_step = ITERATIVE-OPTIMAL(S, n)
 5        if i + depth = count  then
 6              C = C ∪ C_step
 7        else
 8              UC_1 = most profitable group of isomorphic subgraphs in C_step
 9              C = C ∪ UC_1
10              S = {s ∈ S : ∀s' ∈ UC_1,  s ∈ s' = ∅}
```

Fig. 9. An approximate covering algorithm

	4-1		3-2		4-2	
	considered	leaves	considered	leaves	considered	leaves
rawcaudio	5 / 108	1	48 / 885	17	360 / 2 207	125
rawdaudio	149 / 201	464	140 / 696	199	1 293 / 1 958	88
aes	1 112 / 1 233	1	2 749 / 229 127	1	1 112 / 1 233	1
bitcount	6 040 / 9 481	26	4 836 / 18 572	502	44 669 / 86 021	44
blowfish	3 / 75	1	145 / 3 539	29	69 / 3 719	1
des	1 687 / 6 792	4 554	1 781 / 14 232	1	8 933 / 45 579	1
sha	33 / 90	1	37 / 297	1	322 / 703	1

TABLE I
RATIO OF CONSIDERED CANDIDATES TO THE TOTAL NUMBER OF ENUMERATED CANDIDATES, AND NUMBER OF SEARCH GRAPH LEAVES THAT BRANCH-AND-BOUND CANNOT EXCLUDE.

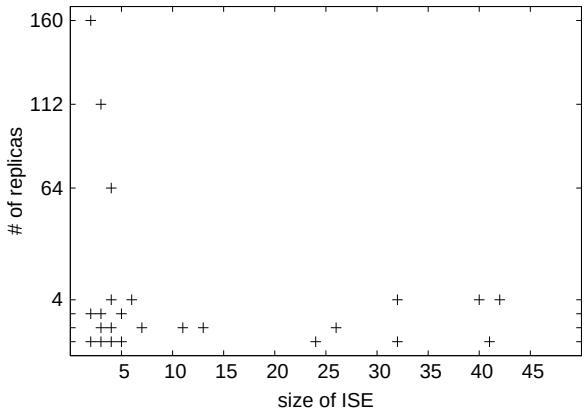


Fig. 10. ISE obtained from the seven benchmarks we used, plotting the instruction's size versus the number of replicas found. (The top half of the y axis uses a different scale).

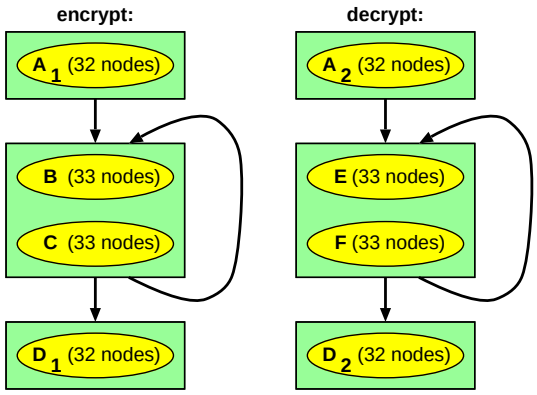


Fig. 11. The entire code for DES encryption fits in four instructions from three basic blocks; likewise for decryption. A_1 and A_2 (resp. D_1 and D_2) are isomorphic and reside in different basic blocks. If their isomorphism is detected, only 6 ISE (instead of 8) are necessary for the hot code to fit entirely into hardware.

A. *Effectiveness of the algorithms*

First of all, we analyze the effectiveness of our proposed algorithms, as well as of recurrence-aware ISE search.

Figure 10 plots the size of the custom instruction found in our suite of seven MiBench programs, versus the number of replicas found for each instruction. For this plot, we ran the enumeration algorithm with a constraint of 4 inputs, 2 outputs.

We can see two effects. The most evident is that in some cases the algorithm was able to find a very high number of replicas for simple ISE—up to 160. These cases correspond to the aes benchmark. In this case, a non-recurrence-aware algorithm cannot find any profitable extension, while interprocedural recurrence-aware techniques can map a considerable number of instructions to application-specific functional units.

We can also see that the algorithm was able to find a few replicas (3 or 4) even for large subgraphs composed of 30 to 40 nodes. In other words, recurrency-aware covering is *not* limited to finding small extensions that, like multiply-accumulate instructions, many processors already implement. This is especially evident for the des benchmark, whose structure is shown in Figure 11. For this benchmark, recurrence-aware search cuts from 8 to 6 the number of instructions that are needed, so that the hot code fits entirely into hardware.

Another point to analyze is the effectiveness of the pruning heuristics. The results are in Table I. For each I/O constraint, the first column gives the ratio between the number of candidates used by the pruned optimal search and the total number of candidates enumerated; the second columns gives the number of leaves explored, that is the number of MIS instances that must be solved. We can see that, especially for the biggest benchmarks only a very small fraction of the candidates is actually explored.

Finally, table II shows compilation times for the greedy, 2- and 3-optimal algorithms. This also shows that the pruning techniques can be very effective: on bitcount, for example, the first exhaustive search examines over 44,000 candidates, yet compilation time grows only by a factor of three between greedy and 2-optimal.

After 2-optimal, compilation time grows much faster, testifying to the complexity of the problem. However, it is important to note that, when the search terminated in less than 8 hours, higher search depths did not improve results over 2-optimal. Only for sha did we measure a small (1%) improvement with 5-optimal search, but in general 2-optimal did a good job with a relatively small cost in terms of compilation time.

B. *Performance results*

Figure 12 shows the results of run-time performance evaluation. The compiler synthesized up to seven instruction set extensions, reaching the limit only for aes, blowfish and sha.

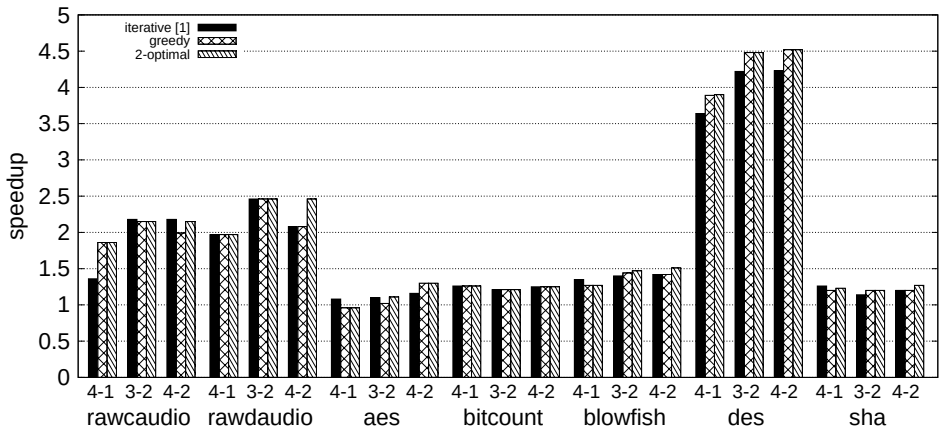


Fig. 12. Speedups obtained with different covering algorithms and I/O constraints.

	greedy	2-optimal	3-optimal
rawcaudio	0.05s	0.08s	40.75s
rawdaudio	0.05s	0.10s	119.65s
aes	9.45s	10.20s	318.60s
bitcount	13.75s	33.10s	—
blowfish	0.07s	0.07s	2.85s
des	2.55s	24.50s	—
sha	0.02s	0.02s	0.02s

TABLE II
COMPILATION TIME FOR THE GREEDY ALGORITHM, AS WELL AS FOR INCREASINGLY ACCURATE APPROXIMATE SEARCHES (4 INPUTS, 2 OUTPUTS).

	Iterative [1]	greedy	2-optimal
rawcaudio	0.00%	7.44%	0.00%
rawdaudio	18.27%	15.45%	0.00%
aes	0.00%	0.00%	0.00%
bitcount	0.80%	0.79%	0.80%
blowfish	0.00%	1.39%	0.00%
des	0.00%	0.00%	0.00%
sha	5.00%	0.00%	0.00%

TABLE III
ADDITIONAL SPEEDUP ACHIEVED BY LOWER CONSTRAINTS, COMPARED TO THE 4-2 CONSTRAINT. NON-ZERO VALUES ARE CAUSED BY THE NON-OPTIMALITY OF THE THREE ALGORITHMS.

Other benchmarks needed fewer than 7 ISE in order to cover all the profitable parts. Even in this case, different complete covers of the benchmarks may have different speedups; indeed this is the case where non-greedy approaches show most potential.

The graph plots the speedups obtained by three different algorithms for different I/O constraints. The first is a version of ITERATIVE search (from [1]), that has been extended in order to find simple cases of recurrences, as described at the end of the previous section. The second and third are algorithms presented in this paper: greedy search (section IV-A), and 2-optimal search (section IV-D). The other two algorithms presented here (exhaustive search, and optimal hybrid search) are used as subroutines of 2-optimal search.

It can be noted that 2-optimal improves on the speedups achieved by the state of the art on non-recurrence aware covering, such as the modified Iterative (first bar). Anomalous behaviour (e.g. aes and blowfish, 4-1) is due to the difference between estimated gains, that search algorithms work on, and actual gains obtained after simulation.

An additional positive property of 2-optimal search is that it is the only one that achieves, for a 4-2 constraint, a speedup that is always comparable or superior to that of lower constraints; this is not necessarily the case for non-exact algorithms. Other algorithms fail to achieve this objective, sometimes substantially (see also Table III). These failures complicate design space exploration, and 2-optimal search achieves most of the benefit of fully exact search not only in terms of speedup, but also in terms of "coherency" in the solution.

VII. CONCLUSIONS

This paper improves on the state of the art on instruction set extensible processors compilation flow, through a range of new algorithms for isomorphism-aware custom instructions selection (*covering*). Besides improving on the speedups achieved by state-of-the-art algorithms, the new techniques exhibited more predictable results as the extended instructions' I/O constraint varied.

The proposed algorithms include optimal search based on branch-and-bound, as well as approximate techniques. These can overcome the obstacle posed by the extremely high asymptotic complexity of the problem, and still achieve a tangible improvement over non-recurrency-aware covering.

REFERENCES

[1] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th Design Automation Conference*, Anaheim, Calif., Jun. 2003, pp. 256–61.

[2] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, San Diego, Calif., Dec. 2003, pp. 129–140.

[3] P. Bonzini and L. Pozzi, "Code transformation strategies for extensible embedded processors," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Seoul, South Korea, Oct. 2006, pp. 242–52.

[4] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2004, pp. 183–89.

[5] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 4, pp. 605–27, Oct. 2002.

[6] N. Clark, A. Hormati, S. Mahlke, and S. Yehia, "Scalable subgraph mapping for acyclic computation accelerators," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Seoul, South Korea, Oct. 2006, pp. 147–157.

[7] P. Bonzini and L. Pozzi, "A retargetable framework for automated discovery of custom instructions," in *Proceedings of the 18th International Conference on Application-specific Systems, Architectures and Processors*, Montreal, Canada, Jul. 2007, pp. 334–41.

[8] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-25, no. 7, pp. 1209–29, Jul. 2006.

[9] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.

[10] A. Peymandoust, L. Pozzi, P. Ienne, and G. D. Micheli, "Automatic instruction set extension and utilization for embedded processors," in *Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors*, The Hague, The Netherlands, Jun. 2003, pp. 103–14.

[11] X. Chen, D. L. Maskell, and Y. Sun, "Fast identification of custom instructions for extensible processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 359–68, Feb. 2007.

[12] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, Apr. 2007, pp. 1331–36.

[13] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction set extensible processors," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Washington, D.C., Sep. 2004, pp. 69–78.

[14] S. Hu and J. E. Smith, "Using dynamic binary translation to fuse dependent instructions," in *Proceedings of the 2004 International Symposium on Code generation and optimization*, San Jose, Calif., Mar. 2004, pp. 213–24.

[15] C. W. Fraser, D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code-generator generator," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 3, pp. 213–26, 1992.

[16] L. Pozzi and P. Ienne, "Exploiting pipelining to relax register-file port constraints of instruction-set extensions," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, San Francisco, Calif., Sep. 2005, pp. 2–10.

[17] A. K. Verma, P. Brisk, and P. Ienne, "Rethinking custom ISE identification: A new processor-agnostic method," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Salzburg, Austria, Oct. 2007, pp. 125–134.

[18] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2006, pp. 212–217.

[19] O. Coudert, "Solving graph optimization problems with ZBDDs," in *Proceedings of the European conference on Design and Test*, Mar. 1997, pp. 224–228.

[20] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," in *Proceedings of the 30th Design Automation Conference*. New York, NY, USA: ACM Press, 1993, pp. 272–277.

[21] H. G. Okuno, S. Minato, and H. Isozaki, "On the properties of combination set operations," *Information Processing Letters*, vol. 66, no. 4, pp. 195–199, May 1998.

[22] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001, pp. 3–14. [Online]. Available: http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf

Paolo Bonzini (S'06) received his M.S. degree in computer engineering from Politecnico di Milano, Milan, Italy, in 2004.

He is currently a Ph.D. student in Informatics at the Faculty of Informatics, University of Lugano, Lugano, Switzerland. His research interests include algorithms for embedded processor customization and compiler optimization.

Laura Pozzi (M'01) received here M.S. and Ph.D. degrees in computer engeneering 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 of 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.

Prof. Pozzi was the recipient of the Best Paper Award in the embedded systems category at the 2003 Design Automation Conference (DAC). She has been a member of the program committee of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) since 2005, of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) since 2006, and was program chair of the Symposium on Application-Specific Processors (SASP) in 2008.