Code Transformation Strategies for Extensible Embedded Processors

Paolo Bonzini, Laura Pozzi
Faculty of Informatics
University of Lugano (USI)
Switzerland
paolo.bonzini@lu.unisi.ch, laura.pozzi@unisi.ch

ABSTRACT

Embedded application requirements, including high performance, low power consumption and fast time to market, are uncommon in the broader domain of general purpose applications. In order to satisfy these demands, chip manufacturers often provide developers with the possibility to define application-specific Instruction Set Extensions (ISEs). Many techniques have been proposed that automatically identify the most beneficial ISEs from source code, so that compilers can identify the 'best' instruction set for the underlying machine. However, can we simply retrofit these techniques into a traditional compiler, or does ISE identification demand different tuning of the heuristics utilized throughout the optimization pipeline? In this paper, we show why compilers should sometimes make different decisions when targeting customized processors, and we show how traditional ISE identification techniques can improve significantly if the code is properly transformed in order to expose more beneficial extensions. The proposed approach was validated using the SimpleScalar simulator for the ARM processor, augmented with the possibility to define additional instructions. Using benchmarks taken from the MiBench suite, we show that the proposed transformations improve state of the art ISE identification techniques by 55% on average and 4x maximum.

Categories and Subject Descriptors: D.3.4 [Processors]: Optimization

General Terms: Algorithms, Performance, Design

Keywords: Customizable processors, ASIPs, Instruction-set extensions, Compilers

1. INTRODUCTION

Processor Customization is an important technique aimed at meeting the stringent requirements of Embedded Processor design: a blend of high performance, low power, and fast time to market that is seldom found outside the embedded applications world. Customizable Processors are quickly becoming available in the market; they are often composed of a standard microprocessor with a simple Instruction Set, that can be augmented with Instruction Set Extensions (ISEs) so that critical parts of the applications can be run in hardware, in application-specific functional units. The automation offered by these processor toolchains is increasing, and source code analysis techniques have been proposed to identify the most profitable ISEs for a given application.

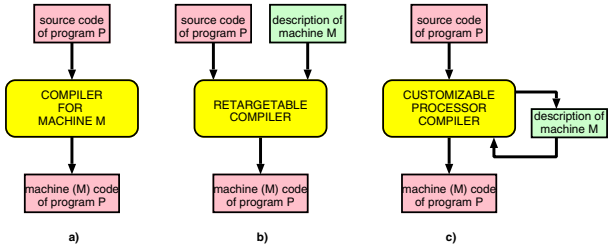


Figure 1: a) A compiler for a specific machine. b) A retargetable compiler reads in a machine description and generates code for it. c) A compiler for a customizable processor can generate the description of the best machine for a given application, and then produce code for it.

Two important observations can be made when looking at this automation trend. The first is that a compiler is taking on a new meaning, as seen in figure 1. Initially, the compiler simply translated a high-level source code into machine code for a given machine, as in figure 1(a). Figure 1(b) shows the role of retargetable compilers that later emerged, and that are able to translate source code into machine code for a set of machines, by reading a machine description as input. We can now go one step beyond and introduce a compiler for a customizable processor—shown in figure 1(c)—that can *automatically generate the machine description*, and then compile onto it (in the case of this paper, decide extensions to the Instruction Set).

A second important observation, central concept in this paper, derives directly from the first: are traditional compiler techniques, aimed at standard (non customizable) microprocessor execution, suitable for this new compilation process, i.e., compilation including the definition of Instruction Set Extensions? Or do traditional techniques need to be redesigned, or at least retuned, in order to be beneficial

CASES'06, October 23–25, 2006, Seoul, Korea.

```
unsigned short
crc (unsigned short crc, unsigned char data)
{
  unsigned char i, x, carry;
  for (i = 0; i < 8; i++)
    {
      x = ((data & 1) ^ ((unsigned char) crc & 1));
      data >>= 1;
      if (x == 1)
        {
          crc ^= 0x4002;
          carry = 1;
        }
      else
        carry = 0;
      crc >>= 1;
      if (carry)
        crc |= 0x8000;
      else
        crc &= 0x7fff;
    }
  return crc;
}
```

a)

```
unsigned short
crc (unsigned short crc, unsigned char data)
{
  unsigned char i, x;
  for (i = 0; i < 8; i++)
    {
      x = (data ^ (unsigned char) crc) & 1;
      data >>= 1;
      if (x)
        {
          crc ^= 0x4002;
          crc >>= 1;
          crc |= 0x8000;
        }
      else
        crc >>= 1;
    }
  return crc;
}
```

b)

Figure 2: a) C code for updating a 16-bit CRC; b) Same code after algebraic simplification, jump threading and value range propagation.

in this new scenario? In this paper we argue that there is indeed a need to revisit traditional compiler transformations, and we notice that current techniques for automated ISE selection can miss significant opportunities that ISE-targeted compiler techniques will expose.

The rest of the paper is organized as follows. Section 2 motivates this research, and section 3 formalizes the problem we want to solve. Section 4 presents the algorithm we implemented and discusses the placement of the proposed techniques in the compiler optimization pipeline. Sections 5 and 6 detail the experimental setup and validate our approach. Related work is discussed in Section 7, while Section 8 concludes this paper.

2. MOTIVATION AND CONTRIBUTIONS

Figure 2(a) shows a C function that updates a 16-bit CRC (using the polynomial $x^{16} + x^{15} + x^2 + 1$) starting from an input byte. This code is extracted from the EEMBC suite's test harness [1]. The compiler can simplify the code a good deal using techniques such as algebraic simplification, jump threading and value range propagation, as shown in figure 2(b). Still, this is not yet a good starting point for ISE search. In fact, ISE identification techniques traditionally operate at the basic block level, with only a few exceptions discussed in section 7, and therefore could identify only small sections in this code as custom instructions. However, this snippet hides very high potential, since ideally the whole CRC computation could be implemented as a custom instruction.

By performing if-conversion *and* total loop unrolling previous to ISE identification, the CRC function is transformed into a single basic block, and state of the art ISE techniques [2] are then able to identify the whole block as a very high-performance 2-inputs 1-output custom instruction. On the other hand, we cannot expect a compiler for a non-customizable processor to aggressively perform such transformations systematically: a traditional compiler is guided by heuristics that limit register pressure and code size increase. But in the case of an IS-extensible processor, since the whole unrolled body can be placed in a single custom instruction, and all intermediate results are transformed into wires, no register pressure and no cache pollution problems arise.

In this paper we propose to tune a set of control-flow transformations which are beneficial for selecting custom ISE. This set is not extensive and can certainly be increased by future research; still it was sufficient to recognize and experimentally prove two important claims: that compilers targeting customizable processors need to be supplemented with new compilation strategies; and that state of the art ISE identification techniques can benefit greatly from the methods here proposed.

3. PROBLEM FORMULATION

The flow typically followed in a compiler that supports ISE is depicted in figure 3(a): an application high-level code is first compiled into a standard intermediate representation (block 1a), which can be seen as a collection of Control Data Flow Graphs (CDFGs). CDFGs capture both control and data flow behavior of an application: for each node (basic block) in the control-flow graph, a data flow graph is associated to it.

Then, an ISE identification algorithm is run on the data-flow graphs (block 2). The problem it solves is formalized in [3]; its output is a set of subgraphs that maximize a merit function—modeling the improvement in execution from hardware implementation of the subgraphs—under imposed microarchitectural constraints (such as the number of

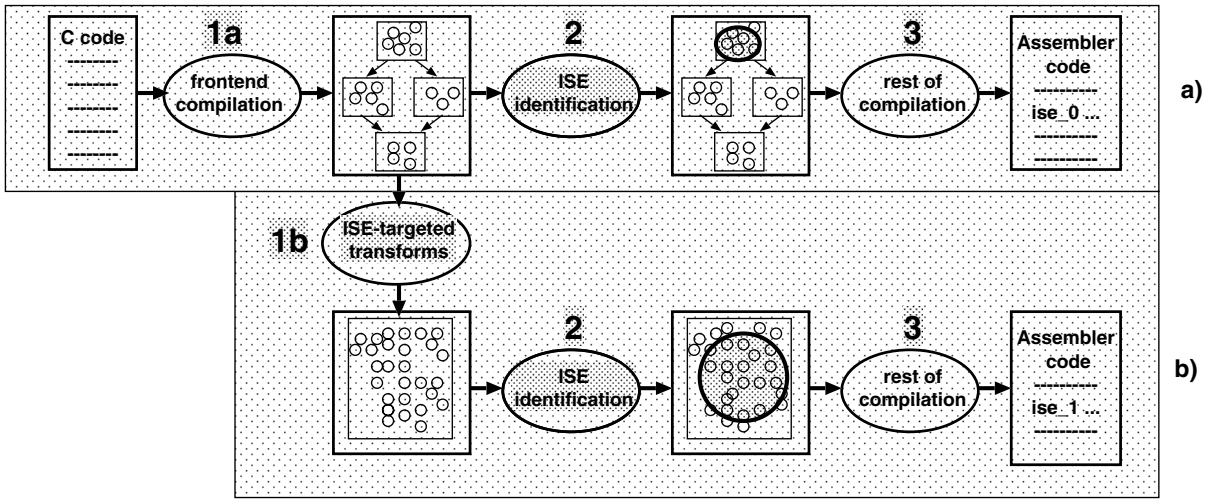


Figure 3: a) Typical flow for ISE identification: an application is first compiled into intermediate representation, then ISE search is performed on it, and then code featuring the new ISE is generated. b) We propose to complete this flow, by applying ISE-targeted transformations to the code, with the aim of exposing better, high-performance ISEs. In this phase, a set of possible CDFG transformations is considered, and the one yielding the best ISE is selected.

read and write ports of a register file). The chosen subgraphs will become the ISEs of the customized processor.

Further on (block 3), all other compilation phases and finally code generation produce assembler code, featuring special opcodes corresponding to the identified ISEs.

In this paper, we *propose to apply a set of ISE-targeted transformations* to the intermediate representation of the application, as shown in Figure 3(b) (block 1b), before ISE identification, in order to *expose* more efficient ISEs. Therefore, we consider an additional phase where the CDFG directly obtained by application source code can be transformed into a semantically equivalent one, but yielding better gain during the subsequent ISE identification phase. In particular, we propose a solution to the problem of *deciding which transformations to apply in a particular instance*, or which semantically equivalent CDFG, among all possible ones, will expose best performance to the ISE phase. Essentially, this problem reduces to defining a transformation space, and choosing a point in it that maximizes ISE potentials.

Before formalizing this problem, we study the meaning of the transformation space, depicted in figure 4. When a set of possible transformation types has been defined (if-conversion and loop unrolling in this example) spots in the intermediate representation can be identified where one of the transformations can potentially be triggered. In the example of figure 4(a), modeled on crc, there are 2 such spots, labeled A and B in the code snippet.

The transformation space can be represented as a tree where every level considers one transformation spot. For transformations types such as if-conversion the choice is binary, and the left branch indicates that the transformation is performed. For loop unrolling, instead, different unrolling factors may also be chosen. The *leaves of the tree* represent all the CDFGs, semantically equivalent to the initial one, that can be obtained by applying the different transformations. The space thus appears to be exponential in the number of transformation spots.

We consider a predefined order for transformation spots to be explored: innermost first, and then in order of appearance in the code. Note that the order of application of transformations does not change the resulting CDFG, for the reduced set of transformation types we have chosen. Studying the effect of transformation types that suffer from phase-ordering is left to future work.

The problem to be solved in block 1b can be stated as follows:

Problem 1 (ISE-targeted code transformation). *Given a graph CDFG and a set of possible transformations to be applied to it in different spots, select the point in the transformation space that maximizes ISE gain, i.e., select the transformed CDFG' that, when fed to an exact ISE identification algorithm, exposes and returns the ISE with highest gain.*

For example, the solutions to the above problem, for applications crc and des, are highlighted with a square in figure 4. These correspond to if-conversion and total unrolling for crc, and to unrolling both loops by a factor of 2 (with no if-conversion) for des. We propose an algorithm for finding a solution to this problem in the next section.

Note that, by describing the problem this way, we do not deal with whether the applied transformations yield the code with the best performance (this can only be verified after compilation and simulation are complete), but only that *they expose the best-gaining ISE to an exact ISE identification algorithm*, a property that can be verified during the immediately following step of ISE identification. This is a very effective way of formulating a problem which is inherently difficult to treat in a general way. In fact, the speedup opportunities that ISE offer are high enough that the best ISE can be assumed to generate the best-performing code as well; possible counterexamples are discussed in section 6.

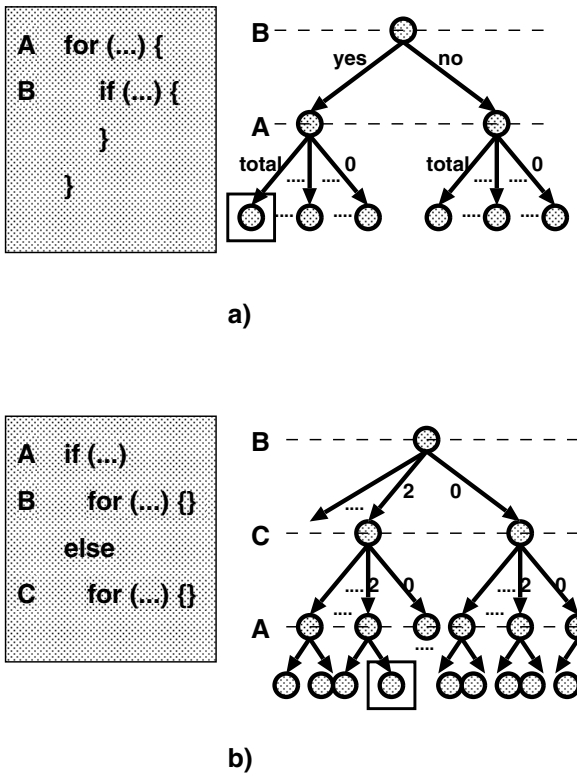


Figure 4: Transformation space, for two examples: crc(a) and des (b). For every spot in the application where a transformation can be triggered (labeled as A and B in the first code snippet, and A, B and C in the second), a decision has to be taken on whether to apply the transformation, and how, e.g. with which factor in the case of unrolling. Each leaf represents a transformed $CDFG$', semantically equivalent to the initial one. Different leaves expose different potential to ISE.

Figure 5 details part of the transformation space for crc (three of the 6 possible leaves), showing how different points in the space expose different potential to ISE. Indeed, ISE identification can find different custom-instructions in each case—depicted as a shaded oval—and each will provide a different cycle saving. The best point, solution to the problem above, is given by the bottom-left $CDFG$', as it contains the best-performing custom-instruction, as anticipated in the motivational example. Note that the transformations leading to it are *not* necessarily the right choice for a compiler which does not target a customizable processor.

4. METHODOLOGY

We propose a solution to problem 1 that considers two compiler transformations: loop unrolling (total and partial) and if-conversion. While these transformations are well known, we propose different heuristics from traditional ones to decide when they should be applied. Indeed, we experimentally show that such strategies *are different from existing ones*, which means that the trivial solution of applying ISE identification at the end of the optimization pipeline does not necessarily yield the best performance.

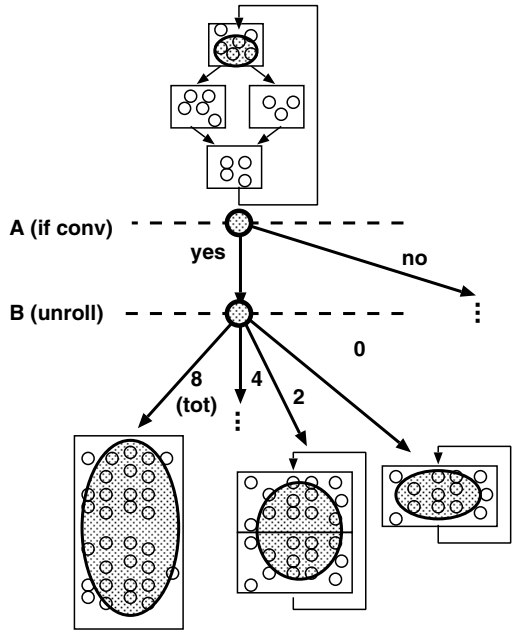


Figure 5: Details of the transformation space for crc, depicting 3 of the 6 possible transformed CDFGs. Each of them exposes a different ISE potential, the best being represented by the if-converted and totally unrolled one.

Of course an obvious solution to problem 1 is to exhaustively explore the transformation space, applying ISE selection to each point, and then selecting the one yielding maximum gain. However this solution is not viable when many transformation spots are considered, and in addition, it has the drawback of possibly applying ISE to points corresponding to huge basic blocks, perhaps resulting from total unrolling. Therefore it is essential to anticipate which points can be eliminated from the search.

In fact, the solution we propose actually applies ISE identification *to a single point only*, and then is able to select, after the single ISE pass, the one point that is the solution to problem 1.

The steps we propose are the following

1. Select a single point in the space, to which ISE identification will later be applied. This corresponds to traversing once the transformation space, taking a decision at each level, in order to reach a leaf. The rules for taking such decisions depend on the transformation type, and are explained in the following subsections.

2. Apply an exact ISE identification algorithm [2] to the $CDFG$' corresponding to the chosen point.

3. Analyze the ISE chosen and identify the transformations actually exploited by it. Select as a winner point the one corresponding to those transformations.

This is better understood by looking at figure 6, based again on the crc motivational example. Step 1 of the algorithm described above selects the leftmost leaf of the tree (indicated with a square) as the one to be fed to ISE identification. This corresponds to an if-converted and totally

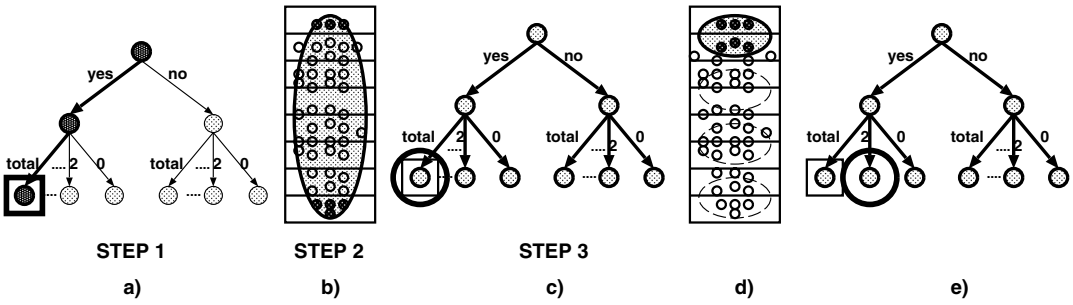


Figure 6: A snapshot of the algorithm, on the crc example. a) In step 1, a leaf is chosen as the single CDFG to which ISE is applied. b) In step 2, an ISE is identified. It contains nodes from first *and* last iteration c) step 3 therefore detects that loop unrolling is exploited, and commits the total-unrolling choice. Since edges to the multiplexer are also included in the ISE, steps 3 detects that if-conversion is also exploited, and commits the choice. As a result, the leftmost leaf is the winner. d) In case the identified ISE only spans 2 iterations instead, e) only an unrolling of 2 is committed and a different point is selected as the winner.

unrolled CDFG. The rules leading to this decision are explained in the following subsections.

Then, ISE identification is applied to it during step 2, and in figure 6(b) it can be seen that the identified ISE spans across the entire basic block. Analysis of the ISE shows that it contains nodes from the first *and* the last iteration. Because of this, total unrolling is committed during step 3: in fact, this signifies that the transformation has been "exploited" by ISE identification. Additionally, the identified ISE includes edges that reach into a multiplexer. Such edges are created by if-conversion; therefore, this transformation was also useful, and is committed. The leftmost leaf, circled in figure 6(c)—the same that ISE identification was run on—is then selected as the winner point.

Figure 6(d) shows a different case, in which the identified ISE spans across first and second iteration only (in this case an isomorphic one is also found across 3-4, 5-6, and 7-8). In this case, an unrolling factor of 2 is enough to expose the best ISE, and a different point, circled in figure 6(e), is selected as the winner.

One important concept in this algorithm is that *performing ISE selection on a single point of the transformation space is sufficient to actually evaluate multiple points.*

In the following, we detail each transformation, and we also show how we determine an upper threshold on the unrolling factor, above which additional ISEs will not be exposed.

Our implementation relies on the CDFGs being in static single assignment form [4]. Each variable is subscripted, and a different subscript is used for each assignment. Additionally, the compiler inserts special definitions at control flow junctions (ϕ functions); these represent different values that the variable can assume, depending on the incoming edge that is followed. This permits a very simple implementation of if-conversion and induction variable analysis.

4.1 If-conversion

An if-conversion pass can be beneficial to the creation of better ISEs, as it can expose additional parallelism and exploit multiplexers in the synthesized functional units. However, unconditional if-conversion can have adverse effects on performance. After if-conversion, the *then* and *else* branches will always execute, and this may induce a greater penalty than removing one or more branches. For a simple, non-superscalar processor, this may happen if the sizes and frequencies of the two branches are heavily skewed: in other words, if one branch is much bigger and also rarely executed.

In fact, even when the architecture supports predicated execution, traditional compilers usually perform if-conversion only if the *then* and *else* branches consist of very few instructions. In our case, the compiler can attempt if-conversion unconditionally, and then roll back the transformation if no ISE will benefit from it.

We can apply if-conversion whenever we find a CDFG region composed of three or four basic blocks and satisfying topology constraints—in this case, the compiler can take a left-branch in the transformation space, during step 1 of the proposed algorithm. The basic blocks must be connected appropriately to represent *if-then* or *if-then-else* constructs, with a *header*, a *junction*, and one or two *conditionally executed* blocks. The junction will be the sole exit of the region; note that (unlike the conditionally executed blocks) the junction may have predecessors coming outside the region[1]. Furthermore, the conditionally executed blocks must not contain any memory access or procedure call, and their outputs must be used only in the junction block's ϕ function.

If-conversion merges the header with the conditional blocks, and possibly with the junction as well. When necessary, outputs in one of the graphs may be merged with inputs in the other graphs. New nodes are created for the junction block's ϕ function, and connected to the conditional blocks' outputs. If an ISE includes a newly created edge, the compiler finalizes the if-conversion by committing it on its SSA-based intermediate representation: that is, the compiler takes a left branch in the transformation space (during step 3 of the algorithm).

Figure 7 details all the steps of this transformation. Figure 7(a) is the CDFG version of the code in figure 2(b). Figure 7(b) shows the single bigger basic block, which is exposed to the ISE selection pass after the transformation. The DFG includes, in the left half, a good candidate for a 3 input, 2 output ISE that includes a multiplexer. Figure 7(c) and (d) show the compiler's internal representation for this

[1]This is acceptable because we are only interested in its ϕ functions, not in its code. We will only examine two ϕ arguments, coming from the other blocks in the if-converted region.

program before and after if-conversion is committed. Here, crc_5 is changed from a ϕ function to a multiplexer, represented by the C language's ternary operator `?:`.

4.2 Unrolling

Unrolling can expose ISE in two ways. A single, disconnected multiple-output ISE can span across multiple iterations, performing them in parallel (we call this an opportunity for *horizontal* unrolling) or, if loop carried dependences exist, a single connected ISE can chain multiple iterations (*vertical* unrolling).

Step 1 of the proposed algorithm requires the compiler to choose an unrolling factor at every level of the transformation tree. This corresponds to a factor that is foreseen to be the maximum one, above which no further ISE potential can be exposed.

For spots that consist of a single basic block, with a self-loop and only one additional outgoing edge, the whole loop body may be covered with a single ISE. The compiler checks if this is possible, and computes the highest value of the unrolling factor n for which this condition holds. The following sets are obtained from the data-flow graph and ϕ functions of the loop body:

H_{in} the inputs that must be provided to the ISE for each iteration. This is the number of operations in the basic block that cannot be computed in hardware: for example, memory accesses or results of function calls.

H_{out} the outputs that the ISE should yield on each iteration. Again, these follow from language characteristics that cannot be mapped to hardware: in this case, values that are passed to subroutines or written back to memory.

V_{in} the inputs that are connected to an output without passing through a node in H_{in}. When the loop is unrolled, only one value every n needs to be passed to the ISE. The ISE can compute the values of these inputs autonomously for the $n-1$ unrolled copies of the loop.

V_{out} the outputs reachable from the inputs in V_{in} without passing through a node in H_{in}. Likewise, the program need not receive the value of these outputs from the ISE, except for iterations n, $2n$, *etc.*

T_{in} the subset of V_{in} nodes that have a constant value at the beginning of the loop. If the loop is totally unrolled, the initial value of these inputs can be hard-coded in the ISE.

T_{out} the subset of V_{out} nodes that are dead at the end of the loop. If the loop is totally unrolled, the final value of these outputs need not be communicated back to the program. For example, the loop index usually contributes to both T_{in} and T_{out}.

For example, the CDFG in figure 7(b) has $H_{in} = H_{out} = \emptyset$ because it does not contain any subroutine call or memory access. V_{in} includes all 3 inputs i, *data* and *crc*. V_{out} includes all 4 outputs (thick-bordered nodes). From figure 7(d) we see that T_{in} contains only i_1, defined by a ϕ function whose value is zero at the beginning of the loop. T_{out} contains i_2, $data_3$ and $exit_1$ which are dead at the end of the loop.

$|H_{in}|$ and $|H_{out}|$ pose a strong limit on the unrolling factor, above which an ISE will not cover all unrolled iterations. Horizontal unrolling puts two or more identical blocks in the same ISE, so that they are executed in parallel. Having $|H_{in}|$ inputs and $|H_{out}|$ outputs on each iterations means that, after unrolling by a factor of n, the ISE would need $n|H_{in}|$ inputs and $n|H_{out}|$ outputs.

Completing this reasoning, we obtain useful inequalities that reduce the transformation space exploration. *These provide an upper limit to the unrolling factor, above which no further benefit can be exposed to the ISE selection pass.*

If a loop is unrolled partially by a factor of n, a hypothetical ISE that covers the whole loop body will have the following number of inputs and outputs:

$$tot_{in} = n|H_{in}| + |V_{in}| \quad (1)$$

$$tot_{out} = n|H_{out}| + |V_{out}| \quad (2)$$

If the loop is totally unrolled, instead, the number of inputs and outputs will be lower:

$$tot_{in} = n|H_{in}| + |V_{in}| - |T_{in}| \quad (3)$$

$$tot_{out} = n|H_{out}| + |V_{out}| - |T_{out}| \quad (4)$$

Now, we transform the equations above into inequalities by noticing that tot_{in} and tot_{out} should not exceed the number of maximum inputs and outputs allowed for an ISE. These inequalities effectively prune the transformation search space, because they limit the number of unrolling factors that need to be explored.

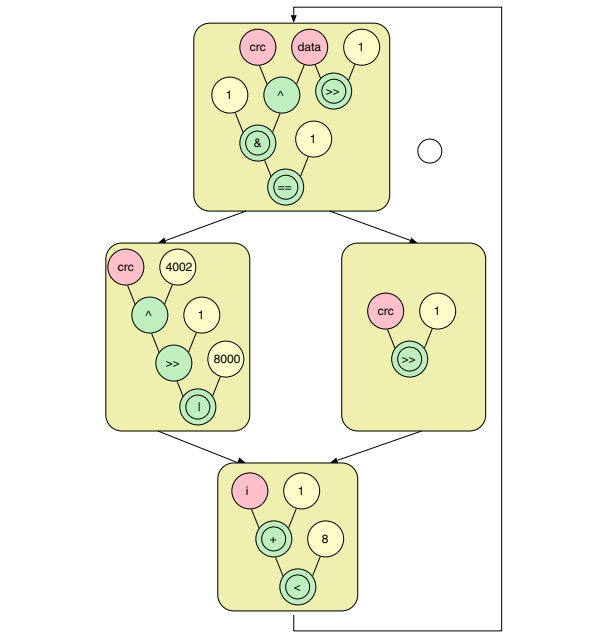
The compiler, during step 1 of the proposed algorithm, will pick the highest integral n that satisfies the above inequalities and that, for a loop rolling a constant number times, divides the number of iterations. If no value of n is a solution, no ISE can cover the whole loop body. Then, the compiler will still unroll the loop by 2, to look for small ISEs across loop iterations—remember that unrolling is not definitive until an ISE is found that can exploit it.

In the case of figure 7(a) we have $tot_{in} = 2$ and $tot_{out} = 1$. This means that the whole loop can be placed into an ISE with an input/output constraint equal to 2-1, and the compiler will perform total unrolling of the loop. If the program includes other code that is hotter, the loop might not be placed in an application-specific functional unit, and unrolling will not be committed in the compiler's intermediate representation. If the ISE is chosen, however, the processor will be able to update the CRC in a single clock-cycle and without any memory access.

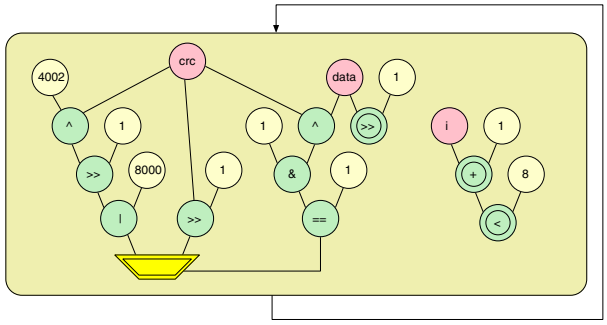
Unrolling is easily performed on the data-flow graph. First, multiple copies of the loop are created and juxtaposed in the same graph. Then, we create edges between each of the copies' outputs and the next copy's inputs. In case of total unrolling, inputs that are constant in the first iteration are hard-coded. Both these operations are easily done by looking up the region's ϕ functions.

5. EXPERIMENTAL SETUP

We implemented our techniques on top of the GCC development trunk (which will become GCC 4.2 in the beginning of 2007), and of the most recent available version of SimpleScalar/ARM. Our extension of SimpleScalar can accept the definitions of up to seven ISEs, each with up to four



a)



b)

⟨bb1⟩
$i_1 = \phi(0, i_2)$
$data_2 = \phi(data_1, data_3)$
$crc_2 = \phi(crc_1, crc_5)$
$x_1 = (data_2 \hat{} crc_2) \& 1$
$data_3 = data_2 >> 1$
if $(x_1 == 1)$ goto ⟨bb2⟩ else goto ⟨bb3⟩
⟨bb2⟩
$crc_3 = ((crc_2 \hat{} 0x4002) >> 1) \mid 0x8000$
goto ⟨bb4⟩
⟨bb3⟩
$crc_4 = crc_2 >> 1$
goto ⟨bb4⟩
⟨bb4⟩
$crc_5 = \phi(crc_3, crc_4)$
$i_2 = i_1 + 1$
$exit_1 = i_2 < 8$
if $(exit_1)$ goto ⟨bb1⟩
⟨bb5⟩
return crc_5

c)

⟨bb1⟩
$i_1 = \phi(0, i_2)$
$data_2 = \phi(data_1, data_3)$
$crc_2 = \phi(crc_1, crc_5)$
$x_1 = (data_2 \hat{} crc_2) \& 1$
$data_3 = data_2 >> 1$
$crc_3 = ((crc_2 \hat{} 0x4002) >> 1) \mid 0x8000$
$crc_4 = crc_2 >> 1$
$crc_5 = (x_1 == 1)\ ?\ crc_3 : crc_4$
$i_2 = i_1 + 1$
$exit_1 = i_2 < 8$
if $(exit_1)$ goto ⟨bb1⟩
⟨bb5⟩
return crc_5

d)

Figure 7: a) CDFG of the crc example after the compiler's scalar optimization passes; b) after if-conversion, the CDFG is replaced by a single basic block; c) SSA representation before if-conversion; d) SSA representation after if-conversion. Nodes with a double border are output nodes.

inputs and up to two outputs; ISE descriptions are written in C and dynamically linked to the simulator. Dually, we augmented the compiler's machine description to match our changes to the simulator.

The compiler includes the ISE identification algorithm called *Iterative*, described in [3] and refined in [2], and uses it to choose the best seven ISEs. It can solve the ISE identification problem for basic blocks of up to around 1000 nodes. Our merit function is an estimate of the speedup; it is obtained from profile-derived execution frequencies, from the instruction count of the subgraph, and from the estimated latency of the ISE hardware implementation. The compiler can include read-only memories in ISEs, as in [5].

After ISE selection, the compiler can also emit C code for the instructions which SimpleScalar will dynamically link to. The tool is able to find multiple occurrences of an ISE within the same basic block, but the identified custom instructions are not reused by GCC in other places in the code. On the other hand, the ISE identification algorithm we adopted pushes for largest, best performing ISEs, regardless of size. Reuse across basic blocks is therefore unusual. This is in contrast with other ISE identification techniques, such as fusion by Tensilica [6], that tend to identify small, reusable ISEs.

The implementation of if-conversion and loop unrolling is based on the Tree-SSA optimization framework. This is available since GCC 4.0 and provides a well tuned set of basic compiler construction blocks which is also relatively easy to use. Tree-SSA supports static single assignment form, alias analysis, and a diverse set of scalar and loop optimizations. These ARM back-end is produces good-quality code and is easy to extend.

We present results for a series of benchmarks taken from the MiBench [7] suites. We selected benchmarks that have

	XScale
Branch predictor	8k bimodal, 2k 4-way BTB
Fetch queue	4 instructions
Fetch/decode width	1 instruction
Issue width	1 instruction, in-order
L1 i-cache	32k, 32-way set-assoc.
L1 d-cache	32k, 32-way set-assoc.
L2 cache	None
Memory bus width	32-bit
Memory latency	12 cycles

Table 1: Configuring SimpleScalar for a popular ARM implementation

	if-conversion	ISE-targeted unrolling	traditional unrolling
rawcaudio	yes	no	no
rawdaudio	yes	no	no
aes	no	yes	yes
aes-table	no	no	yes
blowfish	no	no	yes
des	no	yes	no
sha	no	yes	yes
bitcount	no	yes	yes
crc	yes	yes	yes[a]

[a]On a customizable processor, however, eligible loops are totally unrolled during ISE formation.

Table 2: Transformations that trigger during compilation of the benchmarks. Traditional unrolling may still trigger *after* ISE identification decreases the loops' size.

a small kernel, typically consisting of a single function, and that have little or no initialization code. This helps identifying clearly the speedup that can be obtained from introducing extensions to the instruction set.

We added two benchmarks to those taken from MiBench. One is an AES implementation that, unlike the one in MiBench, uses bit manipulation instead of precomputed tables. The other one is the motivational example in figure 2.

All these benchmarks were compiled with GCC's `-O2` optimization level, enabling feedback-directed optimization and inter-procedural analysis[2]. Note that GCC's usual unrolling heuristics are not disabled: this is because, as previously mentioned, code size reduction due to ISEs may cause them to trigger more often.

We tuned SimpleScalar to match the architecture of the XScale, a commonly used, relatively high-end, ARM implementation. As shown in table 1, this is a single-issue processors with in-order execution. The latency and area of custom instructions were estimated using Artisan UMC $0.18\mu m$ technology. We allowed the introduction of hardware lookup tables in the ISE, that are synthesized from read-only global arrays if it is found profitable.

We considered the processor's cycle time to be the latency of a 32-bit multiply-accumulate cell. ISEs with longer latencies have an execution stage during multiple cycles. The same datasets were used for the profiling run and for the optimized run. Each experiment was run once because the results are deterministic.

[2]These are activated using the GCC options `-fprofile-use -fwhole-program -fipa-cp`.

6. RESULTS

Figure 8 shows the result of running the benchmarks on an augmented XScale. For the first bar, ISEs were identified without transforming the CDFGs, while for the second one, our ISE-targeted transformation strategies were enabled. It can be noticed that the proposed strategies can improve performance tangibly (hatched over solid bar), up to 4x for crc, and 1.55x on average.

In one benchmark only, des, performance is not improved. This is due to the fact that while the ISE identification algorithm used is exact in selecting a single ISE in a single basic block, it is actually greedy in selecting multiple ISEs, i.e. it is greedy in the covering phase. This is further elaborated later in the text.

Table 2 shows the set of transformations that the compiler picks for each benchmark. Encryption benchmarks are especially suited to unrolling, because they are composed of many identical rounds. The motivational example in section 2 shows a performance gain from CDFG transformations close to 4x. As expected, both transformations improve performance on this benchmark. Other tests usually benefit from either if-conversion or loop unrolling, but not both. The inner loops in rawcaudio and rawdaudio span more than one basic block even after if-conversion, and thus are not unrolled; on the other hand, if-conversion gives a substantial improvement. Most crypto benchmarks do not benefit from if-conversion as their inner loops mostly consist of table lookups. If these tables are constant, the compiler will move them inside the ISEs: for this reason, these applications can achieve speedups as high as crc, but only at a substantial area cost.

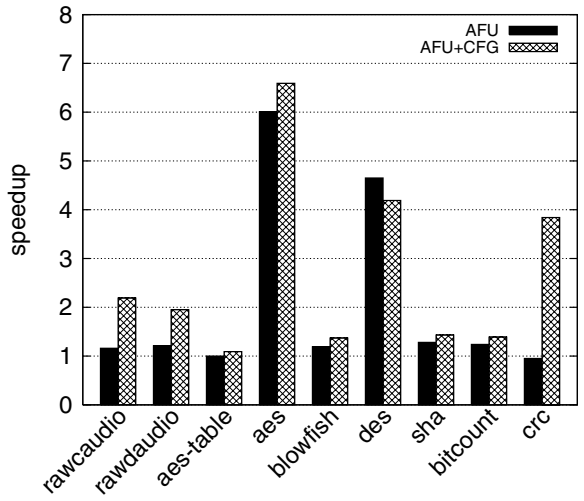


Figure 8: Speedup obtained on customizable processors, without (solid), and with CFG transformations (hatched).

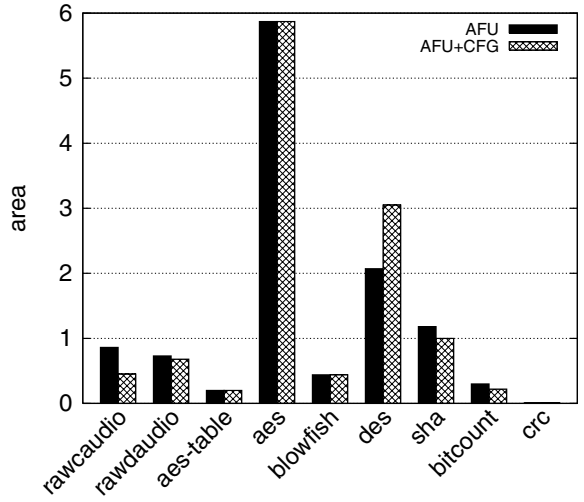


Figure 9: Area cost of the AFUs that were generated without (solid), and with CFG transformations (hatched).

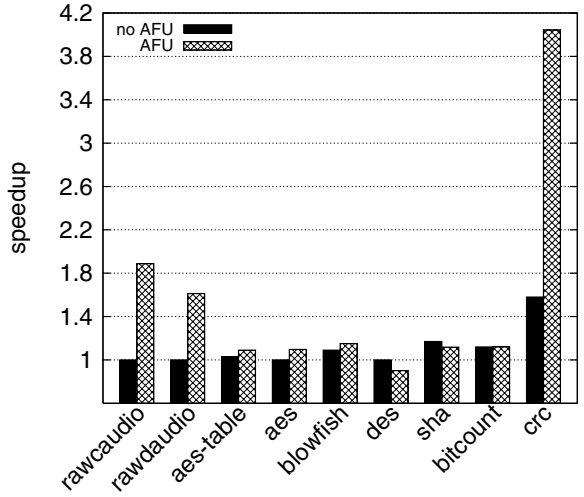


Figure 10: Speedup obtained by CFG transformations, on vanilla versus customizable processors.

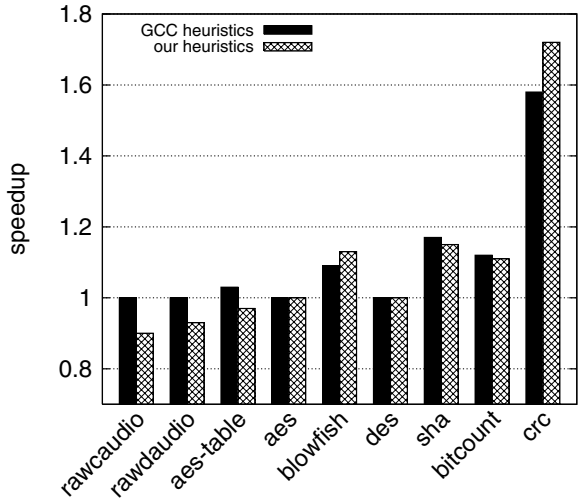


Figure 11: Comparison between GCC's and our heuristics for if-conversion and unrolling, on a non-customizable XScale processor.

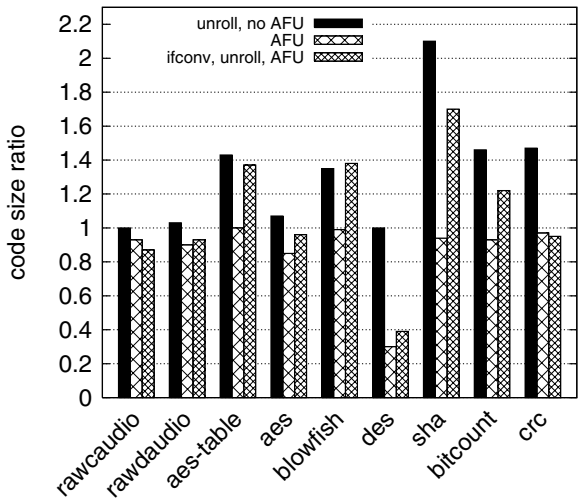


Figure 12: Code size on a non-customizable processor with unrolling, and on a customizable processor without and with CFG transformation.
1.0 = non-customizable processor, no unrolling.

Figure 9 shows the area needed by ISEs for all benchmarks. In most cases, the identified ISEs include ROMs. These can be up to 2 kbit in size, and each have up to four ports. Our toolchain does not take into account area in order to choose the best ISE; however, these figures are probably overestimated because they assume that ROMs are implemented with multiplexers, and do not take into account *block memories* that might be built into reconfigurable fabrics.

Loop unrolling introduces good speedups on benchmarks where it is triggered. des is an exception, with the speedup falling from 4.6x to 4.2x. This is a second-order effect due to the fact that while the ISE identification algorithm used is exact in the selection of a single ISE, it is not exact in the selection of multiple ones. Therefore, even though the chosen unrolling strategy does expose a higher-performance ISE, during the choice of the subsequent ones the limit of seven is met while still leaving the DES *initial permutation* outside the hardware. For more stringent input/output constraints than 4-2, unrolling opportunities are more limited and the performance dip disappears.

Figure 10 compares the speedup that control flow transformations achieve on a customizable processor, with the speedup that they achieve on a standard ARM. For the customizable processor, we take the input/output constraint which performs best, and speedup is computed as the ratio between this and the speed without CDFG transformations. For benchmarks such as aes, the speedups are smaller compared to the 6x leap determined by the introduction of ISEs; but in general, control-flow transformations are a good bet on customized processors.

Also, while not directly related to the techniques we describe, it is interesting that after ISEs are selected the compiler may choose to unroll more loops because ISEs cause code size reduction—sometimes the very loops that were already unrolled by our pass. Indeed, the plots in figure 12 show the code size reduction benefits of an extended instruction set, and how these still hold when CDFG transformations are applied. Of course, unrolling will yield larger code; still, for many benchmarks, the code size improvement from instruction set extensions offsets the greater footprint of unrolled loops.

Compilation time changes are not plotted because they were mostly in the noise. ISE search usually took just a few milliseconds; it took about 40% of the compilation time only for the two AES implementations, since they have large basic blocks where the nodes are interconnected in complex ways. The cost of ISE search did not change substantially on any benchmark when our transformations were enabled.

Finally, figure 11 validates our claim that traditional compiler strategies for standard (non-customizable) processors are not suitable for exposing ISEs, and that there is need for the ISE-aware strategies proposed here. For each benchmark, the transformed CDFGs that maximized customized processor performance (i.e. corresponding to the selected transformation point) were generated, but the code was then compiled for a non-extensible ARM processor; the resulting performance is shown in the second bar. The first bar, instead, shows the performance of code generated by an unmodified, traditional compiler. As it can be seen, our transformation strategies often causes worse performance, showing that compilation strategies that maximize customizable processors performance cannot be easily reconciled with those used in standard processor compilation.

This is due to worse register allocation for sha and bitcount, and to excessive usage of conditional execution for rawcaudio and rawdaudio. In the case of blowfish, our heuristics are better because GCC's choice of unrolling factor causes too many i-cache misses. In the case of crc, the compiler missed the clear opportunity to use conditional execution; this has been reported to the GCC developers.

7. RELATED WORK

The work in this paper touches on two aspects: compiler transformations targeted towards customizable processors, and improved ISE identification. We will analyze related work on both aspects.

Techniques like loop unrolling and if-conversion have been studied extensively in the past, in the context of RISC or VLIW processors [8, 9], in order to increase instruction level parallelism. This work shows that the heuristics that guide them are not always suitable to customizable processors compilation, and it is the first to propose strategies particularly targeted to exposing Instruction Set Extensions.

The problem of ISE identification is that of analyzing application source code in order to find application segments—subgraphs of the basic blocks of the application—that would benefit from being implemented as a custom instruction. In the last decade several methods have been proposed to provide a solution to this problem [10, 6, 2, 11, 12]. However, none of them focuses on how control-flow transformations could *expose better instruction set extensions*. Typically ISE search is run on basic blocks obtained from straightforward front-end compilation of source code. The work presented here, on the other hand, can use any of the methods mentioned above to identify efficient ISEs, after the the intermediate representation is transformed in order to expose the best potentials.

One work [13] represents an exception since it does apply hyperblock formation in the context of automatic generation of hardware accelerators. However, the present work's aims are more ambitious and generally applicable as we formally study the problem of how compiler transformations can beneficially affect ISE generation, and we provide strategies specifically targeted to extensible processors, which differ from those seen up to now.

In [5], inclusion of writable memory onto instruction-set extensions is considered. The method presented here is orthogonal to extensions of that kind, that is, we propose compiler transformations that expose better identification of ISE, and that are also beneficial when memory operations are allowed in the ISE.

[14] suggests a different, more dynamic acceleration device. A Configurable Compute Accelerator can be configured on the fly to implement a sequence of non-accelerated instructions; since the device's interconnections are fixed, there are constraints on the input/output ports, as well as the maximum depth of the implemented subgraph. The authors describe a code motion algorithm that increases the size of the data-flow subgraphs to be executed in hardware. This technique could be applied to CDFGs together with the other transformations we describe in this paper. Dually, our techniques could be applied to a CCA-based architecture.

An important point to consider is the placement of the ISE selection pass within the rest of the compiler. Custom Processor producers such as Tensilica [15] or ARC [16] offer tools that work on the source code and aid the designer in

selecting and implementing extensions to their microprocessors. At the other end of the spectrum, Clark proposes to select ISE near the end of the compiler pipeline [12].

In general, the problem of compilation phase ordering is a significant part of automating ISE design. Just like different phase orderings can have a significant impact on the quality of the compiled code, different placement of ISE selection in the compiler pipeline may result in a very different choice of extended instructions. In our case, the compiler performs ISE selection before all the low-level optimizations that GCC performs on its *RTL* representation.

However, our solution to problem 1 basically extends ISE selection with a pre-pass that includes CFG transformations; this is orthogonal to *when* in the compilation process the selection takes place. Techniques such as [17] can be used to find the optimal sequence of optimization passes for a given user application. Alternatively, [18] employs a probabilistic approach to determine good phase orderings.

8. CONCLUSIONS

In order for a customized processor solution to be a winner, automation at various levels of the design is of the utmost importance. This paper deals with the compilation of embedded applications onto Instruction Set Extensible processor. Compilation takes a new meaning, where the compiler chooses the instruction set of the target processor, then compiles onto it. Given this scenario, we claim that traditional compiler techniques are not sufficient anymore. Instead, ISE-targeted code transformations should be developed to expose better performing custom instructions.

We formalize the problem of ISE-targeted code transformation and propose an efficient solution considering if-conversion and loop unrolling. Cycle accurate results are presented using GCC and SimpleScalar, and show how the benefit of ISE can increase if appropriate control-flow transformations are performed. On a standard XScale microprocessor, the same transformations are often slightly detrimental, demonstrating ISE-targeted strategies to be different from traditional compiler heuristics.

Future work in the area of customizable processors compilation will have to address several open problems. It can be fruitful to extend this technique to other typical compiler optimizations, for example tail duplication or SIMD vectorization. This, however, will complicate the structure of the transformation space, so that our current algorithm may no longer be applicable. Additionally, we have not attempted yet to control the cost of the hardware identified by the ISE identification algorithm. This problem is more important as our techniques increase the possible size of the chosen functional units.

The emerging of customizable processors puts the compiler in a new central role. However, this new perspective poses several challenges to the compiler writer. Addressing these problems will be a key step towards making automated design a viable option.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their useful comments on earlier drafts of this paper.

10. REFERENCES

[1] T. R. Halfhill, "EEMBC releases first benchmarks," *Microprocessor Report*, 1 May 2000.

[2] L. Pozzi, K. Atasu, and P. Ienne, "Optimal 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, July 2006.

[3] 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., June 2003, pp. 256–61.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, October 1991.

[5] 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.

[6] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, San Jose, Calif., Oct. 2003, pp. 137–147.

[7] 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. [Online]. Available: http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf

[8] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*. Gold Coast, Australia: ACM and IEEE Computer Society, 1992, pp. 46–57.

[9] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, Oreg., Dec. 1992, pp. 45–54.

[10] 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.

[11] 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., Sept. 2004, pp. 69–78.

[12] N. Clark, H. Zhong, and S. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–70, Oct. 2005.

[13] T. J. Callahan and J. Wawrzynek, "Instruction level parallelism for reconfigurable computing," in *Field-Programmable Logic: From FPGAs to Computing Paradigm*, ser. Lecture Notes in Computer Science, R. W. Hartenstein and A. Keevallik, Eds., vol. 1482. Berlin: Springer, Aug. 1998, pp. 248–57.

[14] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors." in *Proceedings of the 32st Annual International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[15] T. R. Halfhill, "Tensilica's software makes hardware," *Microprocessor Report*, 23 June 2003.

[16] ——, "ARC Cores encourages "plug-ins"," *Microprocessor Report*, 19 June 2000.

[17] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "ACME: Adaptive Compilation Made Efficient," in *Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM Press, June 2005, pp. 69–77.

[18] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson, "In search of near-optimal optimization phase orderings," in *Proceedings of the 2006 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2006, pp. 83–92.