Compilation and design automation for extensible embedded processors

Doctoral Dissertation submitted to the

Faculty of Informatics of the University of Lugano

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Paolo Bonzini

under the supervision of

prof. Laura Pozzi

May 2009

Dissertation Committee

Dissertation accepted on 27 May 2009

Supervisor	PhD program director
prof. Laura Pozzi	**Fabio Crestani**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Paolo Bonzini
Lugano, 27 May 2009

Abstract

During the last few years, the attention to system-on-chip processors focused on customizability and specializing functional units for particular applications. Such processor extensions can increase performance in domains such as cryptography and DSP, without incurring the power cost of superscalar RISC processors and the complexity of entirely customized integrated circuits.

Since it is extremely desirable that tools automate the design of accelerator units as much as possible, compilation takes on a new extended meaning in this context—the compiler's job is not only to produce optimized assembly code for a machine, but also to *find the optimal machine* for which to compile.

The two different subproblems involved in this task are to automatically *generate* the best instruction-set extensions (ISEs), and to *use* them extensions, i.e. finding them in the user's applications. Therefore, most of the thesis deals with formalizing these combinatorial problems in general, characterizing them for particular architectures, reducing them whenever possible to well known graph optimization tasks, and solving them.

Another wholly different part of compilation for extensible embedded processors is to understand and solve the unique challenges that are posed to a compiler that can actually shape its target machine. In particular, existing compilation techniques can be redesigned and retuned to find even better instruction-set extensions—just like a general purpose compiler will transform the program to maximally exploit the target processor's machine language.

Since this subject exposes many different facets, this thesis touches areas such as compilation, combinatorial optimization, and hardware design. Results presented include: complexity proofs for several algorithms and problems from existing literature; a framework for automated discovery of custom instructions, which is reused in different contexts and for different customization technologies; and techniques for analyzing the effect of compiler optimizations on ISE search. All contributions are prototyped in complete compilation and simulation environments.

Contents

Chapter 1

Introduction

Designing an embedded system entails many choices, and the outcome may often depend on non-technical factors; economic ones such as time-to-market may be especially relevant. Several ways have thus been proposed to mediate between a general-purpose processor's flexibility and ease of use, and a custom integrated circuit's performance in terms of speed and power consumption. One particular solution, which enjoyed support from several vendors, augments the processors with accelerator units accessible through *instruction set extensions* (ISEs) as shown in Figure 1.1).

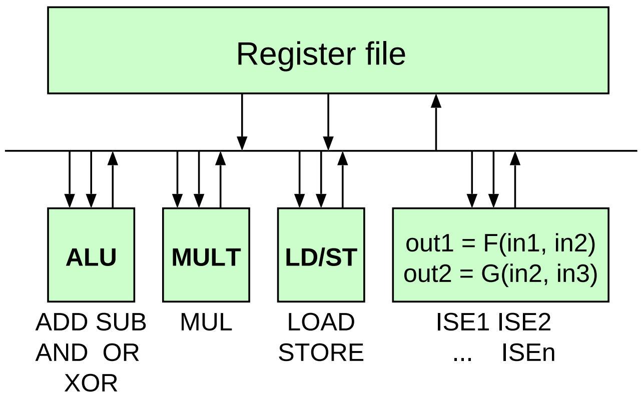


Figure 1.1. A customizable processor includes, in addition to the traditional functional units, one or more *application-specific functional units* (AFU) that can be activated with application-specific *instruction set extensions* (ISE).

Tools to effectively design accelerators for a specific application have not yet enabled complete automation of the process; still, independent of the underlying accelerator technology, automated toolchains are necessary to provide all the supposed benefits of customization. Hence, they are fundamental for wider acceptance of extensible processors. The problem that these tools should solve can be informally stated as follows: *given an application and a target acceleration platform, decide which parts of the code should be mapped onto the accelerator in order to maximize performance*.

This can be called the *mapping problem*; solving it entails effectively searching for good instruction set extensions (*design automation*) and using them in applications (*compilation*), and it is the topic of this thesis. The main claim is that a substantial part of this problem can actually be solved without considering at all the target technology, or at least abstracting it into a small set of parameters or *oracles*; we substantiate this claim by analyzing existing work from the state of the art. The first practical consequence is the definition of a set of subproblems that, together, provide a solution to the mapping problem. We define these subproblems and propose specific interfaces that should be obeyed by the algorithms solving the problems; this allows easy reconciliation of different solution strategies. This set of subproblems and interfaces is called a *solution framework* throughout the work.

The thesis begins with a survey of related work (chapter 2). The chapter explores the customization techniques that have been proposed for embedded processors, as well as the corresponding approaches to compilation for this kind of processor. Incredibly many different architectural solutions for accelerators exist in the literature, and so many different algorithms were conceived to compile on them; therefore, only a small part of which could be presented here. Still, the quality of the comparison with related work for existing papers is sadly very low, so that (to the best of our knowledge) there is no published survey of these techniques.

Section 2.3 then attempts to fill the gap by proposing a *generalization* of most existing research—the aforementioned solution framework. This has two main purposes. First, it may help other researchers filling the blank left by the lack of an extensive comparative survey of the state of the art. Second, it lays grounds for the next chapters of the thesis, in which new solutions for the subproblems are presented, together with new results or improved formulations regarding existing solutions.

The two subproblems are *enumerating* possible instructions ("candidates") and *covering*, which is the process of choosing which candidates will actually be used and where in the program. They are respectively the topic of Chapter 3 and Chapter 4.

The contributions of Chapter 3 include new formulations of the enumeration problem which are more general or more easily understood, as well as new results on existing algorithms. In particular, it shows how some instances of the enumeration problem can be reduced to enumeration of independent sets; this provides an interesting parallel with covering which, in a very broad sense, is an optimization problem on independent sets [Guo et al., 2003]. The chapter also includes the description and experimental results for a very simple yet very fast algorithm for independent set enumeration.

However, the most interesting result of Chapter 3 is the proof that a well known enumeration algorithm [Pozzi et al., 2006] has time complexity polynomial in the basic block size—instead, this algorithm had been wrongly conjectured in the literature as having exponential worst case complexity. The chapter also presents a generalization of this same algorithm which can be used in different scenarios to make the execution even faster.

Chapter 4 formalizes covering in a novel and more general way than what exists in the literature. This new formalization includes the possibility to detect isomorphic candidates interprocedurally, thus avoiding useless reconfiguration or duplication of silicon. New algorithms, both greedy and optimal, are presented that can use isomorphism information effectively and without excessive run-time cost. The chapter's results also show how to go easily from an intraprocedural solution to an interprocedural (whole-program) solution. Finally, the chapter ends with experimental results on the feasibility and gain from isomorphism-aware search, and from optimal (as opposed to greedy) search.

As mentioned before, however, the proposed framework defines not only subproblems to be solved, but also oracles that aid in retargeting the framework to new hardware. An example of this is found in Chapter 5, where the design of a coarse-grained reconfigurable accelerator is presented together with a set of *technology mapping* steps that target the accelerator. Some of these steps can themselves be seen as enumeration and covering problems, and as such they reuse some of the results of Chapter 3.

As a brief aside, the chapter is concluded with a theoretical result, showing that *I/O scheduling*, one of the involved steps (first presented by Pozzi and Ienne, 2005), is NP-complete.

All the steps described so far are already sufficient to construct a compiler for a customizable processor. However, can *something else* be added for increased effectiveness? This is of course true for complex accelerators that target entire loops (as opposed to a set of instructions in a single basic block), on which advanced pipelining techniques are necessary to achieve the best possible speedups; the final chapter of this thesis shows that the framework analyzed so far is by no means the end of the story. Traditional compiler transformations can be revisited and tuned again to produce not only the best possible *code*, but also the best possible *ISE*. The chapter analyzes a case study of two code transformations, if-conversion and loop unrolling, and shows how both can indeed benefit the search for ISE.

Chapter 2

Related work

This chapter surveys existing work on customization of processors, regarding both hardware design (Section 2.1) and compilation (Section 2.2). The goal of this chapter is to present existing work on the topic, spanning almost 15 years, and to classify and evaluate the state-of-the-art.

The outcome of the survey is in Section 2.3, where we propose a framework for compilation to an extensible processor. This framework is adaptable to multiple hardware targets, and is separated in well-defined steps with a clean interface. These steps are then analyzed one by one in the next three chapters.

2.1 Customizable processor hardware

Customization of processors is an alternative to high-level synthesis, in that it can achieve high levels of performance without totally abandoning cheap and accessible software execution. In high-level synthesis, an application-specific circuit is built from the combination of a finite-state machine for control plus a combinational datapath; in a customizable processor instead assembly language replaces the finite-state machine, and the datapath is based on the processor's standard functional unit, augmented with other application-specific units accessible via *instruction set extensions*. Potentially, this can decrease the costs of the final product, by eliminating or limiting non-recurring engineering work of producing custom integrated circuits.

Among the very first examples of instruction set extensions, [Rauscher and Agrawala, 1978] is especially striking. The techniques proposed in the article are based on writable microcode and, while the concept of control storage is almost not found anymore on modern processors, still they anticipate many of the concepts of a modern customizable processor. The authors go as far as proposing automatic generation of microcode—in other words, they define the concept of a compiler for customizable processors:

> *. . . we replace the code generation phase of a compiler by a procedure that [. . .]:*
>
> 1. *generates a microprogram that defines an architecture for efficiently supporting the higher level language program being compiled (this includes interpreting the machine language programs of the newly defined architecture), and*
> 2. *generates the machine language program for the defined architecture that represents the higher level language program being compiled.*

Two main strategies exist for customization of a processor's instruction set. One is to equip the processor with configurable functional units (accelerators) that are programmed separately and are invoked by extended instructions. The second is to make the additional units completely custom and application-specific; this possibility can be implemented either on reconfigurable hardware or on ASIC technology.

2.1.1 Application-specific functional units

Processors sold as *IP cores* are often configurable; characteristics such as caching, availability of an MMU, multiplier/divider timings can be defined by the customer. Some designs may take this customizability one step further, letting user augment the processor with arbitrary functional units written in a hardware description language (HDL). These units will typically implement application-specific computation in order to make them faster. For example, bit manipulations can be sped up greatly because, in a hardware realization, they can often be achieved simply by a bunch of wires.

Reconfigurable hardware is a natural match for such a processor, as an ASIC-based implementation will still bear the cost of fabricating chips. In order to increase the benefits of reconfigurability, Wirthlin and Hutchings's *Dynamic Instruction Set Computer* (DISC, 1995) proposes to exploit FPGA hardware even more via *run-time* reconfiguration. This allows "paging" instruction set extensions in and out of the array, depending on the program's needs. A similar approach is that of *Chimaera*, by Ye et al. [2000], where the processor core is augmented with a *reconfigurable functional unit*. The RFU consists of a small, run-time reconfigurable array of FPGA-like logic blocks, and can host several instruction set extensions.

The possibility to define application-specific functional units is in fact available on commercial FPGA boards, for example those based on Altera's Nios-II processor [Altera Corp., 2002]. However, on boards whose general purpose processor is *not* realized on the FPGA (such as some high-end Xilinx boards; see Xilinx Inc., 2006), there's a high price to pay for reconfigurability and for the interface between the hard-wired register file and the application-specific functional units. For this reason, instruction set extensions are more often used with soft cores or, alternatively, are realized on an ASIC. Using a custom integrated circuit heavily sacrifices flexibility, especially in the view of software upgrades; however this road was followed for example by Tensilica (with the Xtensa processor; see Turley, 1999) and ARC.

Other researchers have suggested using special-purpose reconfigurable hardware to implement a custom datapath for application-specific functional units. While still more or less "fine-grained", the logic blocks of such a fabric has markedly different design goals compared to an FPGA. The Chimaera project for example proposed a custom logic block that includes lookup tables, but has no sequential elements such as flip flops or latches [Hauck et al., 1997]. A more radical approach is that of *Garp* [Hauser and Wawrzynek, 1997], whose logic blocks operate on 2-bit quantities and include two 4-1 multiplexers, a small shifter, a 3-input adder. The Stretch reconfigurable fabric from Tensilica has an even bigger logic block based on 4-bit ALUs [Rupp, 2003].

2.1.2 Accelerating computation using more powerful functional units

Adding a generic programmable functional unit may not be considered, strictly speaking, as producing a *customized* processor, as the instruction set in this case is fixed for each applications. However, the compilation techniques for these two cases are similar; in either case, multiple computations are grouped into a single invocation of either a custom instruction or the accelerator unit. Grouping occurs at the data-flow level, with limited attention to loops and control flow[1].

Razdan and Smith [1994] present an accelerator, the PFU (*Programmable Function Unit*) that is basically a set of small RAMs plus an interconnection that adds more flexibility and allows any input bit to be used as an input for one or

[1]One notable exception is if-conversion (see Section 6.2), which is used to aggregate instructions from different basic blocks and execute them in parallel.

more RAMs. Besides implementing arbitrary sequences of boolean and shift operations, the paper details how the PFU can be used to implement if-conversion (predication). However, the technique is strongly limited by the impossibility to perform arithmetic in the PFU; speedup on SPECint92 is 16% or lower on all but one benchmark.

A more convincing approach is given by the CCA (*Configurable Computation Accelerator*) presented by Clark et al. [2003]. The CCA includes several ALUs organized into rows, and with a flexible interconnect between rows; different rows may support a different set of operations—for example some may support shift operations, others may support adds, and others may support only logical or move operations. The CCA is a relatively big device (it may include a dozen or more ALUs) and has a fixed, multi-cycle delay.

2.1.3 Coarse-grained reconfigurable architectures

Besides FPGAs, several other reconfigurable architectures (CGRAs) have been proposed which are *coarse-grained* [Hartenstein, 2001]. In a CGRA, the structure of the processing element is such that not all boolean functions can be implemented by it—this defines the reconfigurable fabric's *grain*.

However, the definition is broad, and the actual coarseness of the elements differs widely in the design. For example, the cell will usually implement a single arithmetic operation, but is actually a general purpose processor in designs such as RAW [Waingold et al., 1997] or ADRES [Mei et al., 2002]. This makes the respective merits of different coarse-grained architectures hard to assess.

In addition, different approaches have been envisioned also for the CGRA's level of integration in the architecture hierarchy. For example, Morphosys [Singh et al., 2000] uses an 8x8 array of processing elements as a coprocessor, communicating with the processor via FIFO buffers; instead, ADRES has a more tight coupling with the underlying VLIW core and is able to access data from a shared register file.

Coprocessors usually have complex control unit that is able to execute entire loops in a pipelined fashion; designs sitting closer to the processor can instead be regarded as an accelerator, in the fashion described in the previous section. We present such a CGRA-based accelerator design—the Expression-Grained Reconfigurable Array (EGRA)—in Chapter 5. Its main characteristic is a computational element including several ALUs with a flexible interconnect; the most similar example in the state-of-the-art is probably the *Flexible Computational Component* [Galanis et al., 2006] which, while targeted more specifically to DSP kernels, is similar to our processing element in size and set of allowed operations. Unlike Galanis et al., however, our design does not have a fixed structure, but can be tuned to achieve varying levels of complexity and coarseness.

2.2 Extensible processors and compilation technology

Given a customizable architecture, its wide acceptance depends on the availability of tools to effectively design the extensions needed by a specific application. Complete automation, while desirable, is not generally available in vendor-provided tools; these usually aid in the task, but still require the custom instructions to be hand-coded in a HDL.

Nevertheless, a large number of approaches were tried to solve this problem, which can be stated as follows: *given an application and a target acceleration platform, (a) decide which parts of the code should be mapped onto the accelerator in order to maximize performance, and (b) provide an implementation of these parts on the accelerator*. The first part is of particular interest, since the related techniques can often be applied to different kinds of hardware; the general approach followed is to split the process further in two parts, generation of custom instruction candidates and matching them in the program [Kastner et al., 2002; Guo et al., 2003]. These two steps, which will be called *enumeration* and *covering* in this thesis, can be done sequentially [Clark et al., 2006], or they can be iterated [Kastner et al., 2002; Atasu et al., 2003].

The rest of this section will analyze separately two different kinds of solution to this problem. In the case of fully customizable processors, the hardware allows huge flexibility, but the solution becomes more expensive; in the case of accelerators, such as complex functional units or coarse-grained reconfigurable architectures, the obtainable speedups are relatively low, but compilation becomes more affordable.

2.2.1 Compilation techniques for accelerators

Because of the small footprint of accelerators, and of the small program portions that can be mapped onto it, most research on the topic of compilation for accelerators relied on greedy algorithms. Such techniques are simple to implement. Broadly speaking, they start at a “seed” and form small groups of instructions around it satisfying the accelerator’s specifications. Example of such techniques can be found in articles by Clark et al. [2003] or Sun et al. [2004].

Regarding coarse-grained reconfigurable arrays, the challenge of mapping applications on a variety of CGRAs, or in other words to achieve easy retargetability, has never been tackled systematically so far. The reason for this, undoubtedly, lies in the profound differences between the processing elements and interconnect used by different architectures. Most of the previous work concentrated on a single design, and the potential for generalization was rarely pointed out. Two notable exceptions are the work of Lee et al. [2002] regarding loop-level code transformations, and of Chattopadhyay et al. [2008] on architecture description languages for CGRAs.

For example, DRESC [Mei et al., 2002] is a compiler for coarse-grained reconfigurable architectures that performs modulo scheduling and place-and-route for topologies with an arbitrary interconnect. The techniques it uses are designed for the ADRES architecture, but can be used in general for CGRAs employing VLIW processors as functional units. Of particular interest in DRESC is also the representation of the target architecture as a *routing graph* modeling elements such as functional units, register files, multiplexers and buses.

Yoon [2008] and Ahn [2006] studied different approaches for mapping applications onto possibly heterogeneous CGRAs. They provided an ILP formulation of CGRA place-and-route and proposed two efficient algorithms for the same problem: one first groups elements into columns, and then lays out the nodes on the grid [Ahn et al., 2006]; the other uses techniques from planar graph drawing [Yoon et al., 2008]. These strategies are very flexible and can use cells as routing elements—the former only in special cases, while the latter in much more flexible ways.

Guo et al. [2003] present an algorithm to extract templates and find them in an application. This technique is not necessarily related to coarse-grained reconfigurable systems, and is in general applicable to customizable processors. It is equivalent to the *subgraph enumeration* step introduced in Section 2.3 and analyzed in depth in the next chapter.

2.2.2 Compilation techniques for customizable processors

Techniques for customizable processor compilation are in general simpler, but computationally more intensive, than those for accelerators. Only simple constraints have to be satisfied for a subprogram to be executable in hardware, and especially for ASIC technologies it is easy to establish the gain of doing so.

Even very simple formalizations of the problem can actually yield surprisingly good results. This is the case for the work of Atasu et al. [2003], where an algorithm is proposed to find the best performing instruction set extension in a basic block. Despite relying on a greedy methodology to find more than one

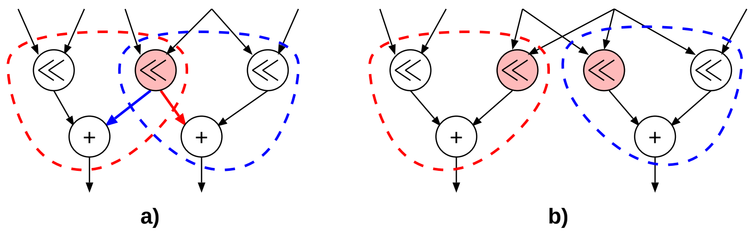


Figure 2.1. Operation duplication in Cong et al. [2004]. a) Only a single output is considered for every chosen subgraph. The shaded node is not considered an output of either dashed subgraph, even though it has edges leaving the subgraphs. b) In order to move both subgraphs to hardware, the shaded left shift has to be duplicated (computed twice).

instruction, and not taking recurrence into account[2], this article has the merit of showing the feasibility of exact solutions to the problem. Many definitions introduced in this paper are used throughout the literature as well as in this thesis (Section 3.1); this is also the algorithm (in the refined version of Pozzi et al., 2006) that we analyze in depth in Section 3.3.

One important classification criterion is the preference for finding very large instructions that can execute very efficiently (this is the case for Atasu et al.), or rather for small program pieces that are found multiple times [Goodwin and Petkov, 2003]. The latter approach can use greedy algorithms such as those for accelerators; it does not achieve very high speedups, but it has the advantage that instruction selection is much simpler for small instruction extensions and can be made optimal [Aho et al., 1988; Liao et al., 1995]. In fact, there is little work on reuse of complex custom instructions, except for special cases such as reuse within a single basic block.

Of course, there are several practical obstacles to an effective implementation. For example, bit-width analysis of the source program can be needed to avoid excessive area usage and to correctly estimate the effectiveness of an hardware implementation [Stephenson et al., 2000; Mahlke et al., 2001]. These were included also in the experimental platform used for this thesis.

In the remainder, we shall highlight a few works that attack the problem in unique ways, touching on aspects that have been otherwise ignored.

Cong et al. [2004] propose a formalization of this problem that exhibits several unique traits. In particular, duplicating the same operation across multiple ISE is not considered in general in the literature, while it occurs naturally us-

[2]Both of these limitations will be tackled in Chapter 4 of this thesis.

ing Cong et al.'s methodology (Figure 2.1). In addition, another desirable property of this approach is that extensions can be chosen so that they obey a given area budget. The main limitation of this work is that it splits the covering phase between candidate selection (driven by area constraints) and matching (driven by performance criteria). This decoupling may result in unused candidates or otherwise non-optimal choices.

Regarding the actual implementation of extensions on silicon, the most interesting publication on this subject is Brisk's [2004]. In this work, the authors show how to build a single datapath for multiple instructions and how to share hardware resources used by different extensions. An apparently similar technique is proposed by Peymandoust et al. [2003], who applies algebraic techniques to find expressions that are subsets of others. The two techniques however have different purposes; while Brisk et al. aims at reducing silicon area, Peymandoust et al.'s technique is used to allow higher reuse of instructions.

Finally, research on customizable processor is closely related to the area of code compression, and the same algorithms can often be used profitably in both applications. For example, recognizing the presence of multiple isomorphic instruction-set extension within a basic block is the same problem as effectively using *echo instructions* [Brisk et al., 2005] or other DISE (dynamic instruction stream editing) techniques [Corliss et al., 2003].

2.3 A framework for customizable processor compilation

This chapter surveyed a body of research in the field of compilation for extensible embedded processors. A natural step is then to classify existing literature according to criteria that can hint at challenging problems and unexplored research in this field. In particular, such criteria may include the following three:

- desired search scope, that is whether the algorithm will look at small parts of the applications (basic blocks) or bigger parts (possibly the entire code);
- desired analysis effort, for example how hard the algorithm will look for similarities between different parts of the application;
- characteristics of the accelerator; this criterion will be left aside momentarily, as we are concerned mostly with the target-independent part of the compiler.

analysis effort →

search scope →

	no isomorphism	exact isomorphism	generalized isomorphism	datapath synthesis
single basic block single subgraph	*exact* Pozzi et al., 2006 Pozzi and Ienne, 2005 Bonzini and Pozzi, 2007a	none	*preliminary* Clark et al., 2003 Peymandoust et al., 2003	*incomplete* Brisk et al., 2004
multiple basic blocks single subgraph	*exact* Pozzi et al., 2006			
single basic block multiple subgraphs	*simple accelerators* Clark et al., 2006			
multiple basic blocks multiple subgraphs	*simple subgraphs* Goodwin and Petkov, 2003 Cong et al., 2004			

Table 2.1. Classifying related work according to search scope and analysis effort. The algorithms presented in this thesis cover the first three columns of the table.

Four different search scopes can be considered, corresponding to search for single or multiple instructions, into one or more basic blocks. We also propose four possible analysis efforts, namely: considering every candidate instruction in isolation; considering all isomorphic candidates together; grouping different candidates into one that subsumes the functionality of all the original candidates; merging different candidates into a single area-efficient datapath. Based on this scheme, past work in this field is summarized in Table 2.1.

As it can be seen, most research concerned the simplest analysis effort, with some exceptions mentioned earlier in this chapter—in particular the work of Brisk et al. [2004] and Peymandoust et al. [2003]. Literature on increasing the search scope is also limited, possibly because Pozzi et al. [2006] showed the intractability of searching for the best instruction set extensions (ISEs) in a global fashion (either intra- or inter-procedurally). However, the problems of a limited scope are real and will be presented in Section 4.4.

In short, past literature proposed solutions that are very valuable, but only solve easier instances of the problem. In addition, the solution were often applicable only to specific instances, and not easily extendible to different cases. The scheme proposed in some cited works is depicted in Figure 2.2(a)(b)(c).

The simplest example, in Figure 2.2(a), splits the job in two phases that execute alternatively. These are searching for potentially useful custom instructions (subgraph search), and choosing, iteratively, the ones that will be actually used in the code (unoptimal covering).

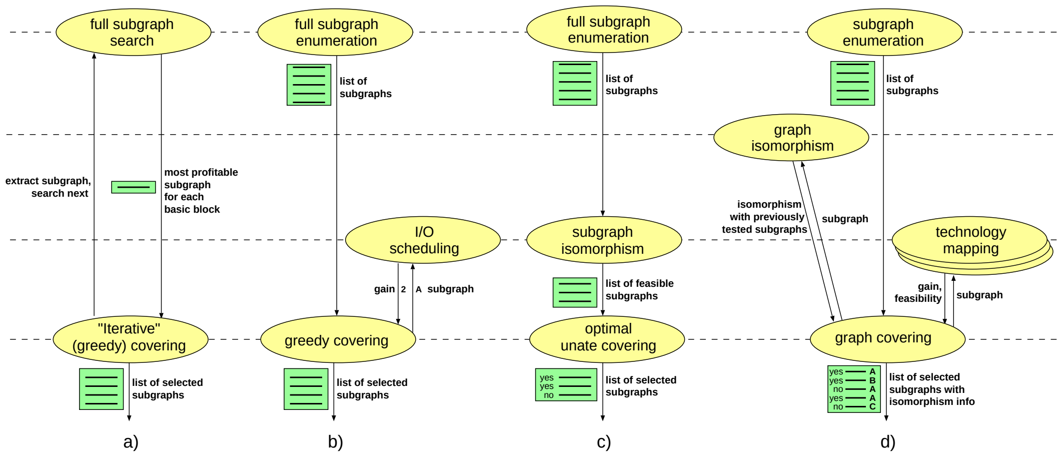

full subgraph search
extract subgraph, search next
most profitable subgraph for each basic block
"Iterative" (greedy) covering
list of selected subgraphs
a)
full subgraph enumeration
list of subgraphs
I/O scheduling
gain
2
A
subgraph
greedy covering
list of selected subgraphs
b)
full subgraph enumeration
list of subgraphs
subgraph isomorphism
list of feasible subgraphs
optimal unate covering
yes
yes
no
list of selected subgraphs
c)
subgraph enumeration
list of subgraphs
graph isomorphism
isomorphism with previously tested subgraphs
subgraph
technology mapping
gain, feasibility
subgraph
graph covering
yes A
yes B
no A
yes A
no C
list of selected subgraphs with isomorphism info
d)

Figure 2.2. Schematic representation of the existing literature, and proposed framework for the subgraph mapping problem. a) The Iterative algorithm [Atasu et al., 2003; Pozzi et al., 2006]. b) Pipelined I/O [Pozzi and Ienne, 2005]. c) Optimal subgraph mapping on a Custom Computing Accelerator (CCA) [Clark et al., 2006]. d) The proposed framework.

Figure 2.2(b) depicts the approach of Pozzi and Ienne [2005] and Verma et al. [2007] to overcome the limit of 2 inputs and 1 output per instruction that is present in most customizable processors. This approach, which we'll examine more closely in Section 5.3, is to add an I/O scheduling phase that readjusts subgraph gains by considering a limited-bandwidth register file—an instance of what we broadly call technology mapping.

The last work depicted, by Clark et al. [2006], also decomposed the problem solution in several phases, all of which were solved optimally; in addition to exact subgraph enumeration [Pozzi et al., 2006], the authors proposed an optimal covering phase based on branch-and-bound.

Seeking to develop a more general problem formulation, we developed a framework for solving the problem of compiling for a particular customizable processor. The framework is shown together with these diagrams in Figure 2.2(d), so that its generality can be appreciated.

Our approach partitions the solution of the mapping problem into two main phases, namely subgraph enumeration and graph covering, and two *oracles* for isomorphism detection and technology mapping. The problem is solved optimally if exact algorithms are employed for all steps—however, heuristics can be used in any of the phases, or might have to be used in some instances because of complexity. This partitioning, together with a clearly specified interface between phases, makes sure that the solutions to the subproblems can be easily reconciled, and makes the framework more directly retargetable to a wider range of accelerators.

The enumeration and covering steps, as well as a technology mapping step, can also be identified in the diagrams of Figure 2.2(b)(c)(d). However, none of them identifies a flexible and retargetable solution. Instead, in our proposal all of the steps, except technology mapping, can be implemented with platform-independent, retargetable algorithms. These algorithms can be parameterized according to target-dependent characteristics such as microarchitectural or area constraints, thus tuning them without changing their implementation. These, together with the presence of a clear interface around the single step that needs to be fully reengineered, makes this framework applicable to many different architectures.

The rest of this section explains the steps in the framework.

Subgraph enumeration reads an application's intermediate representation and generates the set of all potential custom instructions. These candidates are called subgraphs, for consistency with existing literature and because they actually are subgraphs in the preferred intermediate representation for this step (which is presented in Section 3.1).

Section 2.2 of this chapter surveyed several very effective algorithms for subgraph enumeration. Subgraph enumeration algorithm in general do not vary depending on the targeted accelerator. However, as in a retargetable compiler, they should accept the description of the target as an input, and use it to constrain their output to only include valid graphs.

Examples of possible limitations include: the maximum number of inputs or outputs in a subgraph; the maximum length of the dependency chain in the subgraph; the maximum area of the candidate; a set of operations that cannot be mapped onto the accelerator. In some of these cases, the constraint can actually be used to provide asymptotic speedups of the search, as will be shown in Chapter 3.

Isomorphism detection which groups together potential custom instructions that all perform the same function. Unrolled loops, macro expansion, function inlining, generation of addresses for array accesses will all cause the same code to appear multiple times. Therefore, isomorphism detection is an important step for many targets, where it enables hardware reuse.

This thesis does not present any original algorithm for this pass, since state-of-the-art graph isomorphism algorithms can be used effectively. The usage of the oracle in covering is discussed in Chapter 4.

Technology mapping is the phase where target-dependent details are concentrated. Given an input subgraph, it assesses the feasibility of moving it to hardware and the gain of doing so; the covering step (explained later) treats the technology mapping step as an oracle. The accelerator-dependent *subgraph isomorphism* pass of Clark et al. [2006], which is a filter for ISE candidates that cannot be realized on the chosen accelerator, can be seen as a technology mapping step.

Technology mapping can be a challenging problem on its own. For example, precise estimation of the gain might require modulo scheduling and/or place-and-route on some accelerators, such as coarse-grained reconfigurable architectures. Technology mapping often involves scheduling algorithms that are NP-complete (see Chapter 5).

For this reason, the framework includes the possibility of using more than one technology mapping algorithm, each refining the analysis of the previous one. It is then possible to start with a fast mapping algorithm, and only apply more precise techniques to some of the subgraphs.

How technology mapping is used in covering is discussed in Chapter 4; an example of technology mapping is then described in Chapter 5.

Graph covering selects 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 phase as well as the information computed by technology mapping. Covering is formalized, and algorithms to solve it are presented, in Chapter 4.

2.4 Summary

In this chapter, we presented several schemes to enable acceleration of embedded applications, including—but not limited to—the introduction of application-specific functional units. Based on the compilation schemes devised in the past for these technologies, we developed a generic framework for compilation and design automation on extensible processors.

In the following chapters, each step in the framework will be presented in detail, together with novel algorithms and with related experimental results.

Chapter 3

Enumeration algorithms for a customizable processor compiler

In this chapter, we will detail possible solutions for the first phase of the framework presented in Section 2.3. As a foundation that will apply to the next chapters as well, we introduce data-flow graphs and present several definitions that will aid in the description and analysis of algorithms.

We then introduce the most basic enumeration problem (*maximal convex subgraph enumeration*) in Section 3.2, and examine the effects on complexity of the introduction of additional constraints. In Section 3.3 we prove that bounding the number of inputs and outputs of valid subgraphs reduces the problem from exponential to polynomial in the size of the data-flow graph, for any given maximum number of inputs and outputs.

The bound on inputs and outputs, however, is not the only possible one. Section 3.3 defines a framework to deal with more complex constraints such as limited length of the dependency chain.

3.1 Data-flow graphs

The data-flow graph (DFG) of a basic block is an intermediate representation of the statements of the basic block that emphasizes the def-use relationship between statements. A DFG is defined by the set of its nodes V and the set of its edges E. An edge $u \to v$ between two nodes[1] signifies that a value produced by statement u is an input to statement v.

[1]Appendix A describes the notation used in this and the following chapters for graphs (both direct and non-direct) and pseudocode.

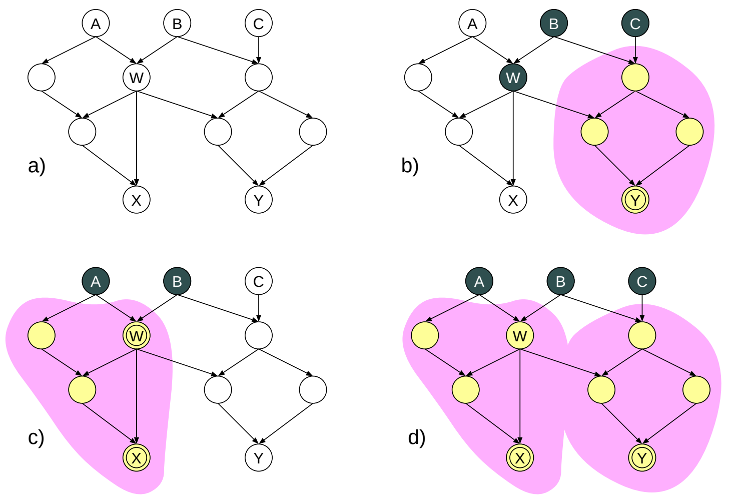


Figure 3.1. a) A simple data-flow graph for a basic block such that W, X and Y are live at the exit of the basic block; b-c-d) Three convex cuts of the graph. Nodes with a double border are outputs and shaded nodes are inputs.

The out-degree of a node can vary, but its in-degree is equal to the number of non-constant operands of the statement. Therefore, once statements have been lowered to expose the desired associativity of operator, data-flow graphs are characterized by a low bound on the in-degree. One exception to this rule is subroutine calls, which can have arbitrarily high in-degree.

The graph G may have an arbitrary number of root nodes I_{ext}, that is nodes that have no predecessors. These nodes represent input variables of the basic block, i.e. variables that are used in the basic block and live at its beginning.

Nodes that are live at the *end* of the basic block are also special and are called *external outputs*; they form another set of nodes, O_{ext}. Assuming dead code elimination has been performed, nodes without a successsor—i.e. whose value is unused within the basic block—are all live at the end of the basic block. Other values however may still not die within the basic block while being used within it. For example, Figure 3.1(a) shows an example data-flow graph with 3 roots (nodes A, B, C) and 2 outputs without successors (X and Y). Remember, however, that X and Y are not necessarily the sole members of O_{ext}; rather, any value that is live at the end of the basic block will be included in it. In the rest of this section we assume that W is live at the end of the basic block.

It is useful to transform G into a single-root, single-sink graph, by augmenting it with a single node that is a predecessor of every node in I_{ext}. We also create an additional node (the *sink*) and connect O_{ext} to the sink. On such a rooted, direct, acyclic graph we can then define the concept of *cut* and in particular of *convex cut*:

Definition 1 (Cut): A cut S is a subgraph of a graph G. The *inputs of S* are the set I of predecessor vertices of those edges which enter the cut S from the rest of the graph G, that is $I = \bigcup_{v \in S} \text{pred}(v) \backslash S$. Similarly, the *outputs of S* are the set O of vertices which are part of S, but have at least one successor $v \notin S$.

Definition 2 (Convex cut): A cut S is convex if there is no path from a node $u \in S$ to another node $v \in S$ which contains a node $w \notin S$.

In the remainder of this chapter, the terms *cut* and *subgraph* (and of course, *convex cut* and *convex subgraph*) will be used interchangeably. The shaded areas in Figure 3.1(b)(c)(d) are all examples of a convex cut. Nodes with a double border are outputs and shaded nodes are inputs; for example the cut of Figure 3.1(b) has inputs $\{B, C, W\}$ and a single output Y.

The objective of enumeration is then to emit a list of convex cuts subject to several constraints. A very generic constraint is to emit only cuts that do not include any node from from a subset $F \subseteq V$ of *forbidden nodes*, that is nodes whose computation cannot be performed on the requested accelerator. These nodes corresponds to subroutine calls and possibly other operations: if the ISE cannot access memory, loads and stores will be forbidden; similarly, floating-point math operations will be marked as forbidden if only integer data can be manipulated by the accelerator. The artificial root and sink nodes are also forbidden.

The remainder of this chapter will examine enumeration algorithms under different sets of constraints.

3.2 Maximal convex subgraph enumeration

First of all, we consider the basic case in which the only requested constraints are convexity and $S \cap F = \emptyset$. Furthermore, we restrict the solution to *maximal* cuts, where adding another node $u \in V \backslash F$ to the cut will violate the convexity constraint; Figure 3.2(a) shows a graph and one of its maximal cuts.

If the solution is restricted to maximal cuts, the problem can be formalized as follows:

Problem 1 (*Maximal convex subgraph enumeration*): Given a direct acyclic graph G with nodes V and edges E, and a set $F \subseteq V$ of *forbidden nodes*, enumerate the maximal elements of the set $\{S : S \text{ is a convex cut of } G \text{ and } S \cap F = \emptyset\}$.

There are two reasons for this choice. First, if all the cuts have to be enumerated, then even very simple graphs will have an exponential-size output[2]. Take for example binary trees: subtrees of a tree are convex cuts, and a tree with n nodes has an exponential number of subtrees [Sloane, 2009, sequence A157679].

Second, problem 1 admits an elegant solution via reduction to a maximal independent set problem. This reduction is interesting for various reasons:

- it shows how the notions of maximum/maximal independent sets, appears throughout different problems in this field and with different definitions of the conflict graph[3];
- it allows to use the efficient algorithm for maximal independent sets of Section 3.2.2;
- it makes it easy to establish the worst case complexity of maximal convex subgraph enumeration.

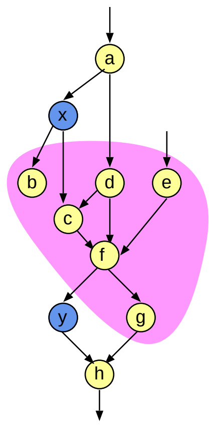
3.2.1 Independent set formulation

Problem 1 was first introduced by Verma et al. [2007]. Verma's solution is based on the definition of a *cluster graph* representing nodes that can stay together in a convex subgraph. Maximal cliques in this graph then correspond to maximal convex subgraphs of the input DFG.

This section will present a formulation of the maximal subgraph enumeration problem that is simpler in two ways. First, we present it as an *independent set* enumeration problem; the relationship to clique enumeration should be clear from the observation that any maximal independent set $S \subset G$ is a clique in the complement graph $\overline{G}$. Independent set formulations are common in the literature (see for example Pothineni et al., 2007 and Guo et al., 2003), because the graph on which independent sets are enumerated (typically called a conflict graph) often has an intuitive definition. This is the case for maximal subgraph enumeration too.

[2]See also Figure 3.11 and Theorem 3 on page 38.

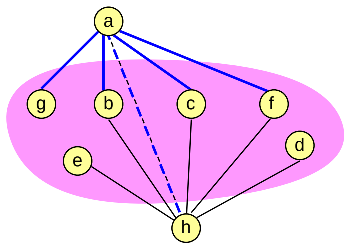
[3]It is common to call a graph a *conflict graph* whenever its coloring or its maximal independent sets have an important meaning for the problem at hand.



a)

node	*FP*	*FS*
a	{}	{x,y}
b	{x}	{}
c	{x}	{y}
d	{}	{y}
e	{}	{y}
f	{x}	{y}
g	{x}	{}
h	{x,y}	{}

b)



c)

Figure 3.2. a) A data-flow graph (dark nodes are forbidden) and a maximal convex cut (others are $\{a,d,e\}$ and $\{g,h\}$; b) The corresponding FP and FS sets; c) The conflict graph and the independent set corresponding to the convex cut. The thick and thin edges form two bipartite subgraphs (Theorem 2).

Second, we omit the *clustering* step of Verma et al., and prove in Section 3.2.2 that this step is equivalent to a node-ordering optimization in the enumeration of maximal independent sets or cliques; it can then be performed separately from the construction of the conflict graph, which is the topic of this section.

Together, these two changes simplify the formulation to the point that it can be described with a single, very simple formula (see Theorem 2 on page 24). In addition, it is easily established from the independent that enumeration of maximal convex subgraphs has the same exponential worst-case tme complexity as enumeration of cliques or maximal independent sets. In other words, neither our reduction nor Verma's are turning a P problem into an *EXPTIME* problem.

The maximal independent set problem is a well known combinatorial problem [Garey and Johnson, 1979] that can be stated as follows:

Problem 2 (*Maximal independent set enumeration*): Given an undirected graph G with nodes V and edges E, enumerate all of its independent sets, i.e. all subsets $S \subseteq V$ such that no edge is contained in S, and that at least one edge is contained in $S \cup \{u\}$ for any $u \notin S$.

We will present a fast algorithm for this problem in Section 3.2.2.

Like Verma et al., we start by defining a relation on pairs of nodes $P \subseteq (G-F)\times(G-F)$, such that $(u,v) \in P$ if and only if there is a valid subgraph of G that includes both u and v. For example, in the graph of Figure 3.2(a), the three nodes b, d and g are part of the shaded maximal convex subgraph; therefore, the three pairs (b,d), (b,g), and (d,g) are—together with many others—all part of P.

Given two nodes u and v such that $(u,v) \in P$, it is possible to form a convex subgraph by including all the paths from u to v in the subgraph. For example, all convex subgraphs including d and g will all include nodes c and f too, because convexity requires inclusion of the two paths $d \to f \to g$ and $d \to c \to f \to g$.

The only case in which $(u,v) \notin P$ is if there is a path between u and v including one forbidden node. In this case, u and v cannot coexist in any convex graph. Again referring to figure 3.2(a), $(a,c) \notin P$ because the forbidden node x lies on the path $a \to x \to c$.

To compute P, we start from the following functions on nodes:

$$\begin{aligned} FP(u) &= \mathrm{Pred}(u) \cap F \\ FS(u) &= \mathrm{Succ}(u) \cap F \end{aligned}$$

representing respectively forbidden predecessors and forbidden successors. These are tabled, for the same example data-flow graph, in Figure 3.2(b).

Then, for each pair of nodes (u,v) there are three possible cases to consider:

- there is no path from u to v, and no path from v to u; then the subgraph $\{u,v\}$ is unconnected but convex. Therefore, both pairs (u,v) and (v,u) are in P. Furthermore, in this case $Pred(u) \cap Succ(v) = \mathrm{Pred}(v) \cap Succ(u) = \emptyset$, hence $FP(u) \cap FS(v) = FP(v) \cap FS(u) = \emptyset$.

- there is a path from u to v and (since G is acyclic) no path from v to u. Paths connecting u and v go through successors of u and predecessors of v:

$$\begin{aligned} (u,v) \in P &\iff FS(u) \cap FP(v) = \emptyset \\ (v,u) \in P &\iff FS(u) \cap FP(v) = \emptyset \end{aligned}$$

  On the other hand, $Pred(u) \cap Succ(v) = \emptyset$, hence $FP(u) \cap FS(v) = \emptyset$.

- there is a path from v to u and no path from u to v. This case is dual to the previous, with paths connecting u and v going through successors of v

and predecessors of u:

$$\begin{aligned}(u,v) \in P &\iff FP(u) \cap FS(v) = \emptyset\\(v,u) \in P &\iff FP(u) \cap FS(v) = \emptyset\end{aligned}$$

On the other hand, $Pred(v) \cap Succ(u) = \emptyset$, hence $FP(v) \cap FS(u) = \emptyset$.

Then, the following definition can be used:

$$(u,v) \in P \iff (FP(u) \cap FS(v)) = \emptyset \wedge (FP(v) \cap FS(u)) = \emptyset \tag{3.1}$$

which is equivalent to the one given by Verma et al.. Note that P is by definition symmetric, so from now we will use (u,v) to represent an unordered pair of nodes.

The definition of equation (3.1) can be reversed to obtain the complement relation C:

$$(u,v) \in C \iff (FP(u) \cap FS(v)) \neq \emptyset \vee (FP(v) \cap FS(u)) \neq \emptyset$$

C is also symmetric, and each pair in it marks the existance of a path between two nodes that includes a forbidden node.

The pairs in C and P can also be used as the edges in a *non-direct* graph; in the case of C the resulting graph is called the *conflict graph*. The non-direct graph in Figure 3.2(c) for example represents the conflict graph for the (direct) data-flow graph of Figure 3.2(a).

We now prove the key property of the complement relation, which allows to turn every instance of Problem 1 into an instance of Problem 2:

Theorem 1: Given a *conflict graph* whose vertices are $G - F$ and whose adjacency matrix is C, a subset S of vertices forms a maximal independent set of the conflict graph if and only if S is a maximal convex subgraph of G.

Proof. Suppose that a convex subgraph S includes nodes $\{s_1, s_2, ..., s_n\}$. Since no two nodes in a convex subgraph conflict, for any two nodes u and v in the subgraph $(u,v) \notin C$ and $(v,u) \notin C$. Then, u and v are not connected in the conflict graph. Therefore, each convex subgraph corresponds to an independent set of the conflict graph.

To establish the converse, and to prove that maximal convex subgraphs correspond to maximal independent sets of the conflict graph, assume that $S = \{s_1, s_2, ..., s_n\}$ is a maximal independent set of the conflict graph, and that the corresponding subgraph in G is not convex. Then S must include two nodes

u and v such that a path connecting u and v includes a node $w \notin S$. Let's assume without lack of generality that the path is $u \to w \to v$.

u and v do not conflict, so $FP(v) \cap FS(u) = \emptyset$. Furthermore, $FS(w) \subseteq FS(u)$ and $FP(w) \subseteq FP(v)$ because u precedes w and w precedes v in a topological order of G's nodes.

Now, given any $w' \in S$, one of these three cases will occur:

- if there is neither a path $w' \to w$ nor a path $w \to w'$, $(w, w') \notin C$.
- if there is a path $w' \to w$, i.e. $w' \in \text{Pred}(w)$, then

$$\begin{array}{rrcll} & FP(v) \cap FS(w') & = & \emptyset & \text{from} \quad (w', v) \notin C \\ \text{hence} & FP(w) \cap FS(w') & = & \emptyset & \\ \text{hence} & (w, w') & \notin & C & \end{array}$$

- if there is a path $w \to w'$, i.e. $w' \in \text{Succ}(w)$, then

$$\begin{array}{rrcll} & FP(w') \cap FS(u) & = & \emptyset & \text{from} \quad (u, w') \notin C \\ \text{hence} & FP(w') \cap FS(w) & = & \emptyset & \\ \text{hence} & (w, w') & \notin & C & \end{array}$$

Since w does not conflict with any $w' \in S$, the hypothesis that S was maximal is contradicted. □

It is also possible to prove a handy construction of the conflict graph:

Theorem 2: $C = \bigcup_{f \in F} \text{Pred}(f) \times \text{Succ}(f)$. That is, the conflict graph is a union of $|F|$ complete bipartite graphs.

Proof. If two nodes $u, v \in G - F$ conflict, there must be a forbidden node f such that there is a path $u \to f \to v$. Then, u is a predecessor of f, v is a successor of f, and $(u, v) \in \text{Pred}(f) \times \text{Succ}(f)$.

The converse is also true. For all $f \in F$, and all $(u, v) \in \text{Pred}(f) \times \text{Succ}(f)$, then $f \in FS(u)$ and $f \in FP(v)$. Then $FS(u) \cap FP(v) \neq \emptyset$, and u and v conflict. □

Finally, it is worth noting that the reduction of the problem to maximal independent set enumeration is not turning a *P* problem to an *EXPTIME* problem. This can be seen in two ways. The first is that *there exist* algorithms that enumerate maximal independent sets in polynomial time per generated set; if one could prove that the number of maximal convex subgraphs is bounded polynomially, the reduction would still have polynomial complexity. The second is

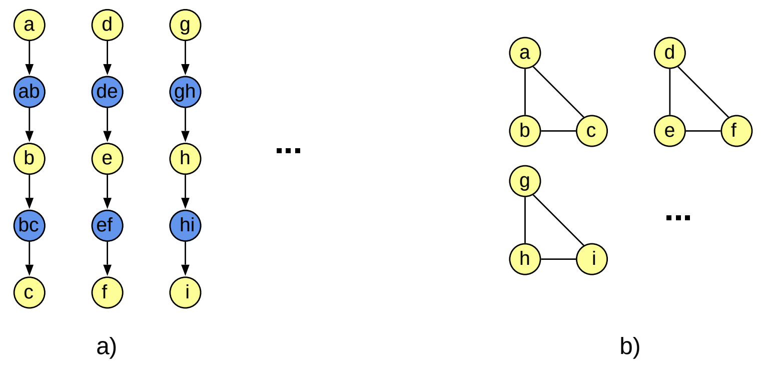


Figure 3.3. a) A graph with $3n$ non-forbidden and $2n$ forbidden nodes. b) The corresponding conflict graph has 3^n maximal independent sets, which is the highest possible number of independent sets for a graph with $3n$ nodes.

that, however, there are indeed graphs with an exponential number of convex subgraphs. In fact, the graph in Figure 3.3(a) has $3^{n/3}$ number of subgraphs where n is the number of non-forbidden nodes in the graph. This is the highest possible number of subgraphs, because in this case the conflict graph—shown in Figure 3.3(b)—is the Moon-Moser graph [Moon and Moser, 1965].

We now proceed to examine how to efficiently enumerate maximal independent sets.

3.2.2 A fast algorithm for maximal independent sets

In the implementation of a fast algorithm for the maximal independent set problem, it is very important to achieve the fastest possible manipulation of the data structures representing the graphs. In particular, nodes that have at least one neighbor in the current solution should be found and excluded as cheaply as possible. For this reason, researchers have proposed to use special representation of the input graph such as *binary decision diagrams* [Coudert, 1997], or special auxiliary data structures that are queried during the execution of the algorithm [Eppstein, 2005].

In this section, we present a fast algorithm to enumerate maximal independent sets on relatively sparse graphs (density $< 70\%$). The algorithm is based on a technique known as *dancing links* that allows fast deletion and reinsertion of nodes into doubly-linked lists—in this case, into the adjacency lists of the graph.

Dancing links were named and popularized by Knuth, who used them to solve another combinatorial problem, the enumeration of *exact covers*[4] [Knuth, 2000].

The technique we present shares the same overall structure as Knuth's *algorithm X* for exact covers. In particular, the core steps are: the deterministic, heuristic choice of a *pivot* in order to limit the branching factor; the nondeterministic choice of a component v of the solution; reduction of the input after including v in the solution; exploration of the subtree. Figure 3.4 specializes these steps to the maximal independent set enumeration problems, and DETERMINISTIC-MIS unfolds the nondeterministic step into backtracking.

However, after unfolding the pseudocode and rewriting the nondeterministic steps into backtracking, the actual algorithms are noticeably different. Even the input, which is in both cases a sparse matrix, has very different characteristics. In the exact cover algorithm, the matrix is rectangular and is the incidence matrix of a hypergraph; in the maximal independent set algorithm it is the adjacency matrix of a graph, and is square symmetric. In addition, while the exact cover algorithm always includes the pivot column in the solution, in our case all but one branches will include *a neighbor* of the pivot rather than the pivot itself.

We will now present the steps of the backtracking algorithm one by one.

Choice of the pivot. The nondeterministic steps of the algorithm include a recursive call, and as such they contribute substantially to the complexity of the algorithm. Each execution of the nondeterministic steps adds a level to a search tree that is implicitly visited depth-first and in preorder.

Whenever different subproblems will lead to the same solution, it is a well known heuristic to prefer branching on the one that minimizes the search tree's branching factor [Golomb and Baumert, 1965]. In our case, this means choosing the vertex which has the fewest neighbors. We can do this easily with cost $O(n)$ by maintaining an edge count for the vertices.

The positive effect of this heuristic on solution speed dwarfs the cost of walking the vertex list at the beginning of each recursive invocation of DETERMINISTIC-MIS. For example, for any undirected graph, if u and v are vertices with the same set of adjacent vertices, and S is a maximal independent set, then

$$x \in S \iff y \in S$$

[4]Interestingly, the optimization version of the exact cover problem is also common in customizable processor compilation; it is the problem called *unate covering* by [2006]; it will also be mentioned later, in Section 4.2.

```
NONDETERMINISTIC-MIS($G$)
    if G has zero vertices, return the current solution
    Deterministically choose a vertex $p$
    Non-deterministically choose a vertex $v \in \{p\} \cup N(p)$
    Include $v$ in the partial solution
    Remove vertices in $N(v)$ from $G$
    Invoke recursively on the reduced graph $G$

DETERMINISTIC-MIS($G$)
    if G has zero vertices, return the current solution
    Deterministically choose a vertex $p$
    for each vertex $v \in \{p\} \cup N(p)$ do
        Include $v$ in the partial solution
        Remove from $G$ the vertices in $N(v)$ and the corresponding edges
        Invoke recursively on the reduced graph $G$
        Add back to $G$ the vertices in $N(v)$ and the corresponding edges
        Remove $v$ from the partial solution
```

Figure 3.4. An algorithm for maximal independent set enumeration

(To prove this, observe that x and y do not have an edge connecting them. Otherwise, y would be adjacent to x and vice versa; the two would not have the same set of adjacent vertices. Then, suppose $x \in S$ and $y \notin S$; since no adjacent vertex of x is part of S, the same can be said for all vertices adjacent to y. Then $\{y\} \cup S$ is also an independent set, contradicting the hypothesis that S in maximal. The same reasoning applies for the case when $x \notin S$ and $y \in S$).

Nodes with the same set of neighbors are common in real-world graphs. In fact, for the particular case of unconstrained maximum subgraph enumeration, this optimization is equivalent to the *clustering* operation that Verma et al. perform as part of the construction of the input graph for clique enumeration. A MIS solver could similarly look for vertices with the same set of neighbors, delete all but one, and merge the two labels. In the case of DANCINGLINKS-MIS, however, this is not necessary, for as soon as one of x and y is added to the solution, the other will have no neighbors and will be included immediately in S.

Graph representation. Since the algorithm does not require any query of the form *"is $u \in N(v)$?"*, the natural representation of the graph uses adjacency lists rather than a bit matrix. In this representation, only the *1* elements of the matrix

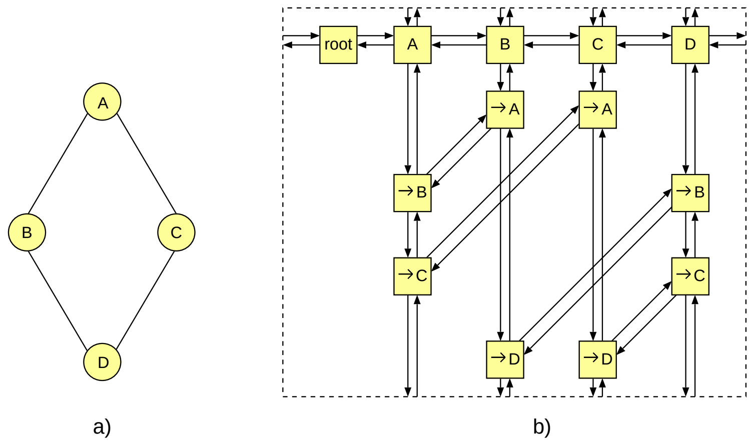


Figure 3.5. a) A simple undirected graph; b) Its in-memory representation using circular linked lists and pointers between dual direct edges.

are connected to a memory object. In addition, there is one such memory object for each column: this *column header* is represented as a dummy edge (u,u), and also includes a "horizontal" linked list of vertices.

The input matrix is symmetric, so each undirected edge (u,v) is represented by two *1* elements corresponding to $u \to v$ and its dual $v \to u$. Instead of providing both same-source and same-destination linked lists, only the former are explicit. However, each edge $u \to v$ has a pointer to its dual, so that same-destination edges can be visited too. As will be apparent later, this solution enables efficient maintenance of the linked lists when vertices are removed and added back to the list.

Figure 3.5 depicts the in-memory representation of a simple graph. The diagram should be seen as wrapping at the four sides, i.e. as a torus.

"Dancing links". The crux of "dancing links" is a simple technique to *put back* removed items of a doubly-linked list. If removing is done with the two operations

$$x.\mathsf{next}.\mathsf{prev} = x.\mathsf{prev}, \qquad x.\mathsf{prev}.\mathsf{next} = x.\mathsf{next} \tag{3.2}$$

then it can be undone with the two other assignments

$$x.\mathsf{next}.\mathsf{prev} = x, \qquad x.\mathsf{prev}.\mathsf{next} = x \tag{3.3}$$

This works because, while $x.\mathsf{prev}$ and $x.\mathsf{next}$ are no longer part of the linked list's chain, they still retain their value and can be used to remember where the

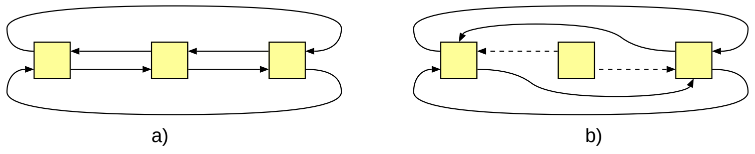


Figure 3.6. Efficiently backtracking node removal on a circular list. It is possible to go from a) to b) and vice versa with only two assignments and no extra storage, provided that all the reinsertions (b → a) are done in reverse order compared to the removals (a → b).

node was placed in the linked list. This is shown graphically in Figure 3.6: the operations of 3.2 go from Figure 3.6(a) to Figure 3.6(b), while 3.3 restores the pointers as in Figure 3.6(a).

Applying this technique to single nodes of the list is trivial; however, it can be extended to sequences of removals and insertions as long as two important invariants hold. First of all, restoring operations must be done in reverse order, so that it can be proved that x.prev and x.next are pointing into the list when they are accessed. Second, there must be a way to reach nodes that are removed this way *even after the actions of (3.2) are applied.* In other words, there must be a way to access the nodes from outside the list that is being manipulated; if x.prev.next and x.next.prev were the only ways to get to x, it would be impossible to restore the pointer during the backtracking phase.

The idea of dancing links–based algorithms, then, is to have two paths to each removed node, and only invalidate one such path during recursive calls. In Knuth's exact cover algorithm, the paths correspond to rows and columns of the matrix. In our case, instead, we only have per-column linked lists, but to those we add (thanks to the symmetry of the input matrix) the links between dual edges such as $u \rightarrow v$ and $v \rightarrow u$.

Removing a vertex v from G entails hiding one row and one column of the adjacency matrix that represents the graph; if the steps of rule (3.2) are used, backtracking can then use rule (3.3). Columns are always accessed by walking the horizontal list embedded in the column headers, so that hiding a column can be done by applying the dancing links technique to the list of column headers; in this case only one node is being hidden (resp. shown), so saving the column header of v into a local variable is enough in order to show it later. However, since the column is hidden, there is no need to hide nodes in the corresponding vertical list: and because the nodes on the removed row are exactly the duals of the nodes in v's vertical list, the dual links and the vertical list together provide the "from outside" path that is needed to apply rule (3.3) when backtracking.

Eliminating duplicates. The algorithm presented in Figure 3.4 has the problem of sometimes finding duplicate independent sets with different orderings. In the case of Figure 3.5, the algorithm processes B first (eliminating A and D and thus finding set $\{B, C\}$) and then C (eliminating the same nodes and finding $\{C, B\}$). To avoid this unfortunate case, we want solutions containing B to be considered invalid while processing C. More generically:

- a total order $\prec$ is established between vertices of the graph;
- all pairs of nodes v_1 and v_2, corresponding to two pivots p_1 and p_2, and such that v_2 is part of both $\{p_1\} \cup N(p_1)$ and $\{p_2\} \cup N(p_2)$, must appear in the solution according to the total order $\prec$ (i.e. v_1 must come before v_2 if $v_1 \prec v_2$).

This test can easily be embedded in the backtracking algorithm DETERMINISTIC-MIS by adding an "invalidity count" to each column header. The final algorithm is presented in Figure 3.7, where the steps left as pseudocode are applications of the dancing links rules (3.2) and (3.3).

Complexity analysis. The space complexity of the algorithm is obviously $O(E+V)$, since the algorithm does not need any data except an $O(V)$ list of vertices, an $O(E)$ sparse-matrix representation of edges, and an $O(V)$ array hosting the current solution.

The time complexity of algorithms for *EXPTIME* problems is usually expressed in terms of asymptotic time per generated solution. For example, Eppstein [2005] proves that his algorithm achieves constant time per generated set on bounded-degree graphs. However, the time complexity of DANCINGLINKS-MIS is difficult to compute for two reasons. First, not all the search tree branches generate a valid independent set, due to the presence of duplicates and to the duplicate elimination test explained earlier in this section. Second, each solution is not generated independently: work done at the higher levels of a tree is amortized across all the independent sets that include that node.

These two phenomena have opposite effects on the complexity. The first one, if ignored, will result in underestimated complexity; the second one instead will cause overestimation. For simplicity, we will ignore both of them. On one hand, these results should not be considered rigorous; on the other hand, we empirically found them to hold well in practice, as we will show.

```
DancingLinks-MIS($G, S$)
    if G has zero vertices, return $S$
    $bestLen = +\infty$
    for each vertex $v$ of $G$ do
        if $|N(v)| < bestLen$ then
            $bestLen = |N(v)|$
            $p = v$
    for each vertex $v \in \{p\} \cup N(p)$ do
        $v$.count = $v$.count + 1
        if $v$.count = 1 then
            Remove $v$ from the horizontal list of column headers
            for each edge $v \to u$ in $v$'s vertical list do
                Remove its dual $u \to v$ from $u$'s vertical list
            DancingLinks-MIS($G, S \cup \{v\}$)
            for each edge $v \to u$ in $v$'s vertical list in reverse order do
                Add back its dual $u \to v$ to $u$'s vertical list
            Add back $v$ to the horizontal list of column headers
    for each vertex $v \in \{p\} \cup N(p)$ do
        $v$.count = $v$.count − 1
```

Figure 3.7. Detailed pseudocode for maximal independent sets enumeration

Each recursive invocation of DancingLinks-MIS has complexity $O\left(deg^2\right)$ where *deg* is the maximum degree of a node in the graph. Since an independent set of size S is built over S recursive invocation of the routine, the complexity is $O\left(S\, deg^2\right)$. This shows that the algorithm is especially suited to sparse graphs. If the degree of every node is the same, then $deg = E/V$ and the complexity becomes $O\left(S\, E^2/V^2\right)$.

Performance measurements. In order to analyze the effectiveness of the algorithm of Figure 3.7, we compared its performance with another state-of-the-art algorithm, the ZDD-based technique described by Coudert [1997]. We consider Zdd-MIS instead of other algorithms such as Eppstein's [2005], because even though it requires specialized data structures and algorithms, these are common in BDD packages such as CUDD [Somenzi, 1998; Mishchenko, 2001] and JDD [Vahidi, 2003].

Note that although enumerating maximal cliques is trivially equivalent to enumerating maximal independent sets (a clique of G is an independent set of the complement graph $\overline{G}$), it is in general unwise to use algorithms designed for one problem to solve the other, because heuristics designed for one problem might not work as well on reductions[5]. In fact, an implementation of the bitmask-based clique enumeration technique described by Knuth [2008, exercises 132–133] did not terminate for most of the sparse graphs considered in this section.

The performance of BDD-based algorithms is often surprising, and their complexity is not well known except for very special cases. Furthermore, they are difficult to tune because of issues such as variable ordering and garbage collection performance. However, ZDDs in practice are less susceptible to these problems [Minato, 1993], making the algorithms of Coudert [1997] a good match for combinatorial problems on graphs.

As we will show, however, the dancing links algorithm beats ZDDs consistently on sparse graphs, and has the additional benefit of achieving optimal space complexity. Its working set has size $O(E+V)$ and will almost always fit in the L2 cache of a modern machine; instead, ZDDs are essentially an application of dynamic programming, and in cases where the computation does not terminate in a short time they would need an extremely high amount of memory.

The graphs in table 3.1 mostly come from toy problems, but are anyway more realistic than random graphs. Furthermore, they make it possible to run the algorithms on the same problem with different graph sizes.

The used graphs are:

***n*-partite** is a 480-vertex complete n-partite graph. This family includes dense graphs, which should be the worse-case behavior of DancingLinks-MIS according to the simple analysis presented earlier; furthermore, it is the optimal case for Zdd-MIS.

***n*-queens** is the attack graph for the queen chess piece in an nxn chessboard. Each vertex corresponds to a square in the chessboard, and there is an edge between two vertices if two queens in the corresponding squares would attack each other.

[5]For example, the column-ordering heuristic of DancingLinks-MIS is able to detect common patterns in graphs and speed up enumeration noticeably. One such pattern is vertices with a single neighbor. If u is such a vertex and its sole neighbor is v, it is possible to delete u, enumerate MIS on the reduced graph, and add back u to all the maximal independent sets that do not include v. Since choosing u yields a branching factor of 2, DancingLinks-MIS will consider u close to the beginning of the search, splitting the search between independent sets including u but not v, and independent sets including v but not u

***n*-rooks** is the same graph for the rook. The number of independent sets, which are all of size n, is $n!$. These two families of graphs were chosen as an example of a graph with small independent sets.

***n*-kings** is the same graph for the king, where the size of independent sets . The size of independent sets varies between $\lceil n/2-1\rceil^2$ and $\lceil n/2\rceil^2$. This family of graphs were chosen as an example of a graph with large independent sets.

***n*-europe, *n*-asia, *n*-euras** are the adjacency graphs for continental European states, continental Asian states, and for continental states in Europe and Asia together.

***n*-usa** is the adjacency graph for successively bigger subsets of the 48 contiguous states of the US (excluding D.C.). n-usa includes the first n states sorted by postal code. These two families of graphs were chosen as examples of bounded-degree graphs, and have large independent sets (they are also planar).

aes is a real-world example of maximal subgraph enumeration, consisting of three different basic blocks, where accesses to non-constant memory are considered as forbidden. This is the only application we found that took a measurable amount of time, and did terminate in a reasonable time.

Table 3.1 shows a performance comparison between the two chosen algorithms. DancingLinks-MIS is faster on all benchmarks except (as expected) for the very dense n-partite graphs. Table 3.2 compares the actual complexity measured on the chessboard graphs with the predicted complexity $O\left(S\,deg^2\right)$. This shows that the negative effect of enumerating duplicated independent sets is in practice negligible, and that the actual complexity is usually lower.

Summary. In this section, we presented a fast and elegant algorithm to enumerate maximal independent sets of a graph, and particularly of a sparse graph.

The algorithm is able to discover characteristics of the graph thanks to a vertex ordering heuristic, but does not attempt to identify special structures of a graph, such as disconnected components or articulation points. Such structures cause combinatorial explosion of the number of independent sets of the graph, and can be problematic for algorithms that (like DancingLinks-MIS) simply do brute-force enumeration.

graph	nodes	edges	# sets	DancingLinks-MIS	Zdd-MIS
2-partite	480	57 600	2	2.83s	<0.01s
3-partite	480	76 800	3	5.19s	<0.01s
4-partite	480	86 400	4	8.06s	<0.01s
8-partite	480	100 800	8	10.50s	<0.01s
12-partite	480	105 600	12	11.40s	<0.01s
20-partite	480	108 000	20	12.30s	<0.01s
30-partite	480	111 360	30	12.50s	<0.01s
40-partite	480	112 320	40	12.90s	<0.01s
48-partite	480	112 800	48	13.10s	<0.01s
8-rook	64	448	40 320	0.23s	0.53s
9-rook	81	648	362 880	2.28s	4.73s
10-rook	100	900	3 628 800	24.20s	52.10s
8-queen	64	728	10 188	0.02s	0.33s
9-queen	81	1 056	57 600	0.14s	3.26s
10-queen	100	1 470	376 692	0.98s	31.50s
11-queen	121	1 980	2 640 422	7.20s	518.40s
7-king	49	156	201 611	0.11s	0.28s
8-king	64	210	6 214 593	3.78s	9.97s
9-king	81	272	391 918 650	230.30s	—
32-usa	32	41	7 770	<0.01s	0.16s
40-usa	40	73	53 246	0.04s	1.19s
48-usa	48	105	219 062	0.18s	15.60s
36-asia	36	56	1 336	<0.01s	0.05s
47-europe	47	86	29 397	0.02s	0.87s
83-euras	83	149	29 275 464	16.50s	803.00s
aes-1	67	452	2 041	0.14s	0.26s
aes-2	69	480	2 289	0.17s	0.34s
aes-3	356	44 896	15 601	1.29s	3.97s

Table 3.1. Comparison of DancingLinks-MIS and Zdd-MIS

			complexity per enumerated set	
family	ind. set size	max. degree	theoretical	actual
n-queen	$O(n)$	$O(n)$	$O(n^3)$	$O(n)$
n-rook	$O(n)$	$O(n)$	$O(n^3)$	$O(n)$
n-king	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$

Table 3.2. Worst-case and actual complexity of DancingLinks-MIS for three families of graphs.

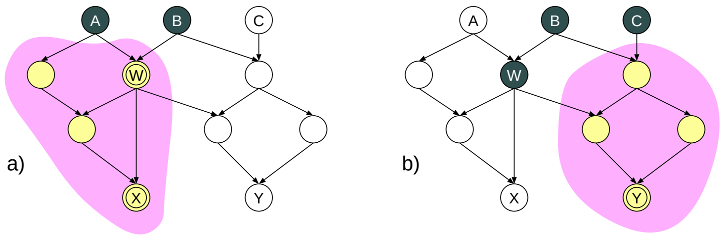


Figure 3.8. Two convex cuts; nodes with a double border are outputs and shaded nodes are inputs. a) A cut with two inputs and two outputs; b) Three inputs and one outputs.

In our experiments, the real-world example of convex subgraph enumeration md5 did not terminate in 24 hours with DANCINGLINKS-MIS[6]. While DANCINGLINKS-MIS was running, it enumerated almost 10^{11} independent sets; however, because of the presence of disconnected components, it had only explored a small fraction of the solution space. Similarly, maximal independent set enumeration took several hours for the graph of the continental states of Europe, Asia and Africa, due to the presence of an articulation point (Egypt).

In these cases, the search space could be split in multiple subspaces, explored one by one with DANCINGLINKS-MIS. Partial solutions can then be expressed effectively as a ZDD and then combined using the basic combination-set operators described by Minato [1993], thus reconciling these two approaches to combinatorial enumeration.

3.3 I/O constrained enumeration

One possible way to make enumeration faster is to reduce the number of enumerated subgraphs by introducing additional validity constraints. At least in a naïve implementation of a custom instruction, there is a direct relationship between the number of inputs and/or outputs in a subgraph and the number of read/write ports in the register file. Therefore, limiting the number of inputs and/or outputs in a subgraph has always been considered by the research community as an “obvious” way to cap the complexity of subgraph enumeration. This is a very strong limitation, which would discard even very simple graphs such as those of Figure 3.8.

[6]It also had to be killed after a few minutes for ZDD-MIS due to excessive memory usage.

Alippi et al. [1999] present an interesting result considering one output but still allowing an unlimited number of inputs. In this case, it is possible to perform enumeration in a single $O(n)$ step, because the maximal subgraphs are disjoint. This also helps in covering, which can be performed greedily on disjoint subgraphs (see also Section 4.2). Unfortunately, this effective solution does not extend well to more general cases where both the inputs and outputs are limited.

Limiting both the inputs and the outputs leads to the following problem:

Problem 3 (*I/O constrained subgraph enumeration*): Given a direct acyclic graph G, a set of forbidden nodes F, and a maximum number of inputs N_{in} and of outputs N_{out}, enumerate all the convex cuts $S \subseteq G$ under the constraints that $|\mathrm{I}(S)| \leq N_{\text{in}}$, $|\mathrm{O}(S)| \leq N_{\text{out}}$, and $S \cap F = \emptyset$.

The problem can be solved efficiently by combining multiple branch-and-bound strategies. This approach was presented by Atasu et al. first [2003] and later refined by Pozzi et al. [2006]. Since these are applied on top of an exhaustive enumeration—i.e. the algorithm branches twice on every node, trying all cuts that include it and all cuts that exclude it—the complexity is trivially $O(2^n)$. It is however interesting to see whether there exist better lower bounds, possibly polynomial in the size n of the graph. In fact, Pozzi et al., as well as Chen et al. [2007], observe that in practice the run-time of the algorithm grows slowly with n, and much faster with the limit in the number of inputs and outputs.

Chen et al. [2007] even prove that the *output size* of Problem 3 is polynomial in n; the polynomial complexity would hence be achieved by a trivial algorithm enumerating all $N_{\text{in}} + N_{\text{out}}$-uples of nodes, and checking whether they satisfy the convexity criterion and have indeed N_{out} outputs[7]. Such an algorithm, however, would not be practical [Bonzini and Pozzi, 2006a].

The proof we present in this section, indeed, is the first that shows that a fast algorithm for Problem 3 has polynomial time complexity in n.

Algorithm. As a first step in the proof, we provide the details of a fast implementation of the algorithm outlined in Pozzi et al. [2006]. While both Atasu et al. [2003] and Pozzi et al. [2006] only overviewed how to implement the branch-and-bound criteria in an efficient manner, examining the details is necessary to prove the algorithm's complexity.

[7]For example, in Figure 3.1(b) of page 18 the cut with inputs (A, B) and output X has an additional internal output W; Bonzini and Pozzi [2006a] detail how this fact can actually be used to optimize the search.

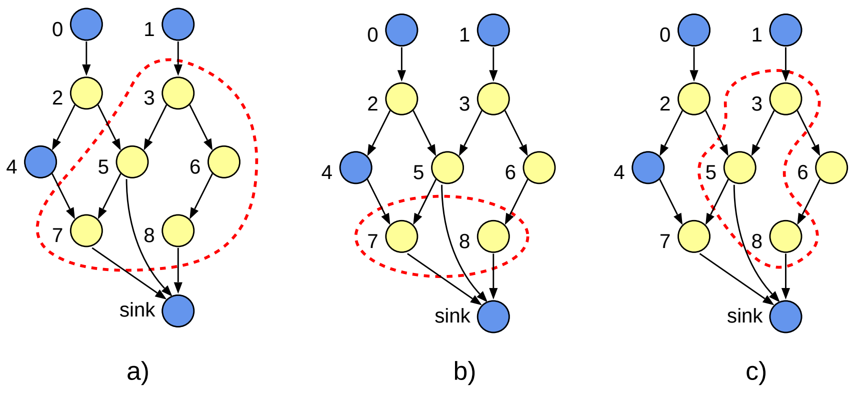


Figure 3.9. Eliminating invalid cuts. Supposing $N_{\text{in}} = 2$, $N_{\text{out}} = 2$, and that the search has reached node 2 (in reverse topological order, i.e. starting from node 8), these three cuts are all rejected: a) has three outputs (5-7-8), b) has three permanent inputs (4-5-6), c) is not convex. Entire branches of the search tree can be skipped: the cuts are invalid independent of whether or not node 2 is part of the cut.

The basic idea of the algorithm is to order nodes topologically and process them backwards; this allows several optimizations because the following invariants hold[8]:

1. Adding to a convex cut S a node u that (in the chosen topological order) comes before every node $v \in S$, will not remove any output from S.

2. Adding to a convex cut S a node u that (in the chosen topological order) comes before every node $v \in S$, will not remove from $\mathrm{I}(S)$ the inputs coming after u in the topological order. These inputs are called *permanent*.

3. Adding to a non-convex cut a node u that (in the chosen topological order) comes before every node $v \in S$, will not restore the convexity of the cut.

From each of these invariants we can derive a condition that, if broken, allows to discard the entire search tree under S. These conditions, represented in Figure 3.9, are respectively that $|O(S)| > N_{\text{out}}$, that $|\{u_i \in I(S) : i \geq \mathit{index}\}| > N_{\text{in}}$, and that S is not convex.

The pseudocode in Figure 3.10 is an implementation of the algorithm. Function Search tries adding to S all the nodes $\{u_i \in V : i < \mathit{index}\}$, and expects as

[8]Proofs are included in Atasu et al. [2003] (invariants 1 and 3) and Pozzi et al. [2006].

```
SEARCH($S, index, n_{\text{permin}}, n_{\text{out}}$)
    ▷ $u_0$ to $u_{|G|-1}$ represent nodes of $G$, ordered topologically
    ▷ $u_{|G|}$ is the artificial sink node
    ▷ The search is started with SEARCH($\emptyset, |G|, 0, 0$).
    if $u_{index} \notin S$ then
        if $\exists v \in \text{succ}(u_{index}) : v \in S$ then
            $n_{\text{permin}} = n_{\text{permin}} + 1$
        if $n_{\text{permin}} > N_{\text{in}}$ then return
    else
        if $u_{index} \in F$ then return
        if $\neg\forall v \in \text{succ}(u_{index}) : v \in S$ then
            $n_{\text{out}} = n_{\text{out}} + 1$
        if $n_{\text{out}} > N_{\text{out}}$ then return
        if $S$ is not convex then return
        if $|\text{I}(S)| \leq N_{\text{in}}$ then $S$ is a valid cut

    if $index > 0$ then
        SEARCH($S \cup \{u_{index-1}\}, index - 1, n_{\text{permin}}, n_{\text{out}}$)
        SEARCH($S, index - 1, n_{\text{permin}}, n_{\text{out}}$)
```

Figure 3.10. Subgraph enumeration algorithm from Pozzi et al. [2006].

a precondition that S does not include any of them. It also assumes that the last node in the topological order is forbidden. This is true because the last node in the order will have no successors and, after the artificial sink node v_{sink} is added, it will be the only node without a successor.

Our $O(1)$ implementation of search tree pruning works as follows. Function SEARCH maintains a count of outputs and *permanent inputs*, i.e. nodes that are inputs for all the cuts in the nodes that will be explored recursively; these are those inputs $u_i \in \text{I}(S)$ such that $i \geq index$. These two counts are updated on every recursive call. If the number of outputs or permanent inputs exceeds, respectively, N_{out} or N_{in}, exploration of an entire branch of the search tree can be avoided.

Note that of the three invariants, the third is only needed to prove the correctness of the algorithm; the first two instead are also central to proving its complexity. In fact, we can prove the following theorem:

Theorem 3: The I/O constrained enumeration algorithm of Atasu et al. [2003] has exponential worst-case time complexity.

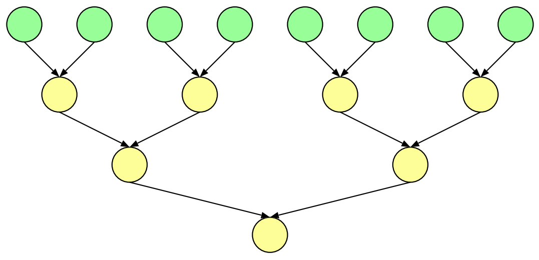


Figure 3.11. A complete n-node tree is a data-flow graph with an exponential number of convex subgraphs.

Proof. The only difference between this algorithm and the one of Pozzi et al. [2006] is that the former does not consider condition 2, while the latter does. Then, consider a data-flow graph G which is actually an upside-down complete binary tree, with n nodes none of which is forbidden (see Figure 3.11). This graph has only one *maximal* subgraph enumeration, but Problem 3 requires all subgraphs satisfying the constraints to be enumerated.

Since condition 2 is not considered, the algorithm behaves in the same way, and has the same complexity, independent of N_{in}'s value. The algorithm will compute the same set of subgraphs for $N_{\text{in}} = 1$ or $N_{\text{in}} = n$, even though most of them will obviously be discarded if N_{in} is low.

For high enough N_{in}, any subtree of G is a valid subgraph. The number of such subtrees is exponential in the number of nodes [Sloane, 2009, sequence A157679; see also sequences A004019 and A115590]. This proves the theorem. □

The same reasoning applies if invariant 1 is removed.

Complexity analysis. Based on this implementation, we will now prove that, in addition to the exponential $O(2^n)$ upper bound for time, this algorithm also admits an alternative bound of $O\left(n^{N_{\text{in}}+N_{\text{out}}}\tau(n)\right)$, where $\tau(n)$ is the complexity of processing a leaf of the search tree and of the convexity test (whichever is more expensive). Our proof is constructive; we transform the pseudocode so that the different upper bound is clearly visible. Still, all the versions of the pseudocode have the same complexity; no further optimizations are introduced.

The first step is to move to the caller the update of n_{permin} and n_{out} according to how many successors of u_{index} are in the cut. The modified pseudocode of Figure 3.13 shows that the three cases of Figure 3.12 are possible:

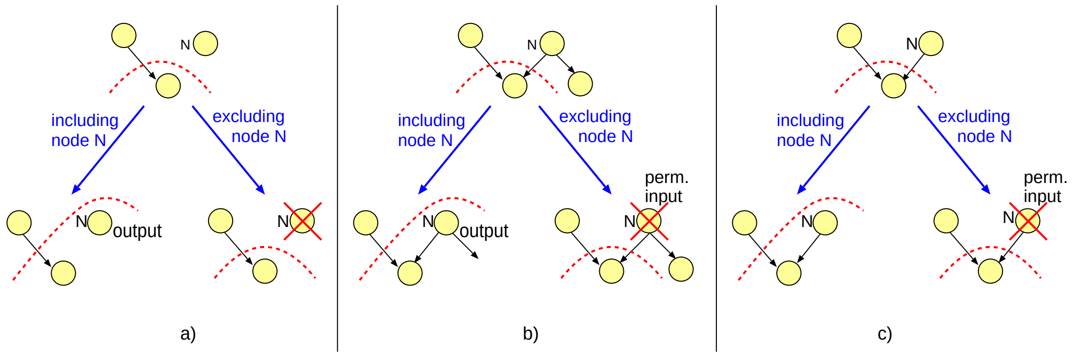


Figure 3.12. Updating n_{permin} and n_{out} after N is included/excluded from the current cut. a) Node N has zero successors in the cut; b) node N has at least one successor in the cut and at least one successor not in the cut; c) node N's successors are all part of the cut.

- if the node has zero successors in the cut, adding it to the cut will create an output;
- if the node has at least one successor in the cut, and at least one successor *not* in the cut, adding it to the cut will create an output, and excluding it will turn it into a permanent input;
- if the node's successors are all part of the cut, adding it to the cut will not create an output, but excluding it will still create a permanent input.

In order to query how many successors of any node are part of S, a side table is updated every time nodes are added and removed from the cut. When node u is added or removed, the count changes for all its predecessor, giving a cost of $O(d_{in})$, where d_{in} is the maximum in-degree of G, for each recursive call. This cost is smaller than $\tau(n)$, because the convexity test can also be done in $O(d_{in})$ time, and thus can be ignored.

We then proceed to transform one of the two recursive calls into iteration. In Figure 3.14, each recursive call then increments one of n_{permin} or n_{out}. Since $n_{\text{permin}} < N_{\text{in}}$ and $n_{\text{out}} < N_{\text{out}}$, there can be no more than $N_{\text{in}} + N_{\text{out}}$ recursive calls active at any time, each of which will execute the **while** loop at most n times. This proves the complexity result given at the beginning of this section.

Pozzi et al. [2006] actually describe a more general condition for declaring an input permanent. In addition to all inputs coming after u_{index} in the topological

```
SEARCH-2($S, index, n_{\text{permin}}, n_{\text{out}}$)
    if $n_{\text{permin}} > N_{\text{in}} \vee n_{\text{out}} > N_{\text{out}}$ then return
    if $u_{index} \in S$ then
        if $u_{index} \in F$ then return
        if $S$ is not convex then return
        if $|\mathrm{I}(S)| \leq N_{\text{in}}$ then $S$ is a valid cut

    if $index > 0$ then
        if $\neg\exists v \in \mathrm{succ}(u_{index-1}) : v \in S$ then
            ▷ No successors are in the cut
            SEARCH-2($S \cup \{u_{index-1}\}, index - 1, n_{\text{permin}}, n_{\text{out}} + 1$)
            SEARCH-2($S, index - 1, n_{\text{permin}}, n_{\text{out}}$)
        elseif $\neg\forall v \in \mathrm{succ}(u_{index-1}) : v \in S$ then
            ▷ Some (but not all) successors are in the cut
            SEARCH-2($S \cup \{u_{index-1}\}, index - 1, n_{\text{permin}}, n_{\text{out}} + 1$)
            SEARCH-2($S, index - 1, n_{\text{permin}} + 1, n_{\text{out}}$)
        else
            ▷ All successors are in the cut
            SEARCH-2($S \cup \{u_{index-1}\}, index - 1, n_{\text{permin}}, n_{\text{out}}$)
            SEARCH-2($S, index - 1, n_{\text{permin}} + 1, n_{\text{out}}$)
```

Figure 3.13. Moving checks to the caller.

order, *all forbidden inputs* (including external inputs I_{ext}) *are permanent*. Since they cannot be included in the cut, adding nodes to S will not remove forbidden inputs from $\mathrm{I}(S)$. This allows the algorithm to achieve even better complexity in practice.

Adding this more efficient condition to our implementation is easy. For the pseudocode in Figure 3.13, for example, it suffices to add the following line at the very beginning of the function:

$$n_{\text{permin}} = n_{\text{permin}} + |(\mathrm{I}(S) \setminus \mathrm{I}(S \setminus \{u_{index}\})) \cap F|$$

Constraining enumeration with data-flow analysis. Earlier in this section, we showed how I/O constraints can be advantageous and can provide lower complexity bounds. We now further extend this observation, presenting a framework to design algorithms according to specific enumeration constraints, with even better performance.

```
SEARCH-3($S, index, n_{\text{permin}}, n_{\text{out}}$)
    $start = index$
    while $index \geq 0 \land n_{\text{permin}} \leq N_{\text{in}} \land n_{\text{out}} \leq N_{\text{out}} \land$
            $\land S \cap F = \emptyset \land S$ is convex do
        if $u_{index} \in S \land |\mathrm{I}(S)| \leq N_{\text{in}}$ then $S$ is a valid cut

        $index = index - 1$
        if $index > 0$ then
            if $\neg\exists v \in \text{succ}(u_{index}) : v \in S$ then
                SEARCH-3($S \cup \{u_{index}\}, index, n_{\text{permin}}, n_{\text{out}} + 1$)
            elseif $\neg\forall v \in \text{succ}(u_{index}) : v \in S$ then
                SEARCH-3($S \cup \{u_{index}\}, index, n_{\text{permin}}, n_{\text{out}} + 1$)
                $n_{\text{permin}} = n_{\text{permin}} + 1$
            else
                SEARCH-3($S, index, n_{\text{permin}} + 1, n_{\text{out}}$)
                $S = S \cup \{u_{index}\}$

    $S = S \setminus \{u_i : i < start\}$
```

Figure 3.14. Eliminating one recursive call.

An example is given by the task of enumerating valid subgraphs for a CCA (*Configurable Computation Accelerator*; see Clark et al., 2006). Besides the number of inputs and outputs, in this case valid subgraphs have to be subgraphs *also* of a "template" data-flow graph representing the capabilities of the CCA.

A conservative approximation of this additional constraint is to limit the maximum depth of valid subgraphs, so that it does not exceed the maximum depth of the accelerator. This leaves precise filtering of invalid subgraphs to a separate check, as done by Clark et al. [2006] (see also Figure 2.2), but the technique presented here however performs subgraph enumeration faster, and runs subgraph isomorphism fewer times.

The maximum depth poses a strong bound on the size of the output. Assuming that the maximum depth is N_{rows} and the in-degree of the data-flow graph is limited by d, the number of valid subgraphs is at most $d^{N_{rows}} n^{N_{\text{out}}}$ subgraphs. This is considerably smaller than $n^{N_{\text{in}}+N_{\text{out}}}$, since d is usually 2 or 3 and N_{rows} is ≤ 7 [Clark et al., 2004]; this means that the solution of Problem 3 includes many graphs (the very large ones) that are of no interest. In fact, Clark reported enumeration times well over 10 minutes for large basic blocks [Clark et al., 2006].

```
DF-SEARCH($S, index, n_{\text{permin}}, n_{\text{out}}$)
    ▷ $u_0$ to $u_{|G|-1}$ represent nodes of $G$, ordered topologically
    ▷ $u_{|G|}$ is the artificial *sink* node
    ▷ The search is started with SEARCH($\emptyset, |G|, 0, 0$).
    ▷ $OUT[v]$ and $IN[v]$ are the propagated data-flow values for $v$.
    ▷ $\wedge$ and $f(\cdot)$ define the data-flow analysis semilattice
    ▷ $limit(\cdot)$ defines the data-flow characteristics of a node
    if $u_{index} \notin S$ then
        if $\exists v \in \text{succ}(u_{index}) : v \in S$ then
            $n_{\text{permin}} = n_{\text{permin}} + 1$
        if $n_{\text{permin}} > N_{\text{in}}$ then return
    else
        if $u_{index} \in F$ then return
        $OUT[v] = \bigwedge_{v \in S \cap \text{succ}(u_{index})} f(IN[v])$
        $IN[v] = OUT[v] \wedge limit(v)$
        if $IN[v] = \perp$ then return
        if $\neg \forall v \in \text{succ}(u_{index}) : v \in S$ then
            $n_{\text{out}} = n_{\text{out}} + 1$
        if $n_{\text{out}} > N_{\text{out}}$ then return
        if $S$ is not convex then return
        if $|\mathrm{I}(S)| \leq N_{\text{in}}$ then $S$ *is a valid cut*

    if $index > 0$ then
        DF-SEARCH($S \cup \{u_{index-1}\}, index - 1, n_{\text{permin}}, n_{\text{out}}$)
        DF-SEARCH($S, index - 1, n_{\text{permin}}, n_{\text{out}}$)
```

Figure 3.15. Data-flow-based constrained enumeration algorithm.

We will now show how to define this problem using a data-flow analysis framework [Aho et al., 1988]. This makes it easy to extend the algorithm to more complex cases.

This is a backwards data-flow analysis problem—for example, the depth of a node depends on the depth of all successors. As common in data-flow analysis, the propagation of values is described by a semilattice and a transfer function, which is called $f(\cdot)$ in the pseudocode of Figure 3.15. A node is only added to the subgraph if its associated value is not the bottom element $\perp$ of the semilattice.

The valid depths of a node in a subgraph, for example can be represented by a semilattice on the set $\{0, 1, \ldots, N_{rows}\}$; the *meet* operator is $a \wedge b = \min(a, b)$, the *minimum* of the two depths, so that 0 is the bottom element $\perp$ and N_{rows} is the top element $\top$. The transfer function is $f(x) = x - 1$; note that the transfer function need not be defined on $\perp$, because it is only applied to nodes that are part of the subgraph.

The data-flow framework includes another function to cater for more complicated constraints. This is $limit(\cdot)$ and it associates an element of the semilattice to each node; it is used to specify the data-flow characteristics of each node. In this simple example, it is always equal to $\top$ (the neutral element of the meet operator), since nodes can be placed at any level.

The solution of the problem has to be recomputed for every candidate subgraph; this is not a problem because the data-flow graph is acyclic, hence the solution can be computed in a single reverse topological order visit. In fact, each node can be visited as it is added to the graph, as in Figure 3.15, thus embedding the solution directly in the enumeration algorithm.

A complete application of the data-flow framework will be shown in Section 5.2.2. As we will show in that section, data-flow-based enumeration listed valid subgraphs of large basic blocks, under constraints similar to the ones presented in this section, in less than a millisecond.

Chapter 4

Covering algorithms for a customizable processor compiler

If enumeration gives the answer to the question "what can be used as a customizable instruction?", covering is the process of deciding "what *will* be used as a customizable instruction" and where. The purpose of this step is to select a set of custom instructions to be actually implemented and used.

Covering is an optimization problem. Therefore, the solution of the problem is *the* set of instruction set extensions with the highest merit. While we used the term *gain* for the outcome of technology mapping passes, *merit* is computed by the covering algorithm *based on* the gain—for example it may include profiling information to favor frequently executed instructions. In this chapter, we will compare different approaches to covering with different complexities, ranging from greedy to *NP*-complete.

Section 4.1 formalizes the problem of covering including the detection of subgraphs that are *isomorphic,* that is perform the same function. This is an important addition to the covering pass, because unrolled loops, macro expansion, function inlining, generation of addresses for array accesses will all cause the same code to appear across multiple basic blocks; detecting this is necessary in order to achieve good speedups with only a few instruction set extensions.

After this introductory step, starting from Section 4.2 the case of non-overlapping custom instructions is analyzed in depth. First the simplest cases of covering are solved. These pose no limit on the number of instruction set extensions chosen, force the merit of a subgraph to be computed upfront, and do not consider the presence of multiple isomorphic copies of the same subgraph.

Then, in Section 4.3 the case of a bounded number of custom instructions is treated, together with two orthogonal extensions to the basic problem. Multiple technology-mapping steps can be added, allowing fast conservative estimates of an instruction's merit to be refined in successive steps. Alternatively, isomorphism can be taken into account; the approach presented in this chapter is able to detect isomorphism across basic blocks or even procedures, and it can favor smaller instructions with higher reusability while retaining the ability to find sizable (30–40 nodes) instructions.

4.1 Problem formulation

In this section, we show a generic formulation of covering. The inputs to covering are:

- a set of graphs, $G = \{G_1, G_2, ..., G_i\}$, representing the application's basic blocks.
- 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 $\bigcup_{G_i \in G} S_i$ of candidate subgraphs taken from all basic blocks, i.e. the set of all candidates in the application.
- a function $M_C(\cdot)$ that takes a subset of S, representing the occurrences of custom instructions that were selected for hardware execution, and return the merit of the choice (for example an estimate of saved clock cycles).

and the problem to be solved can be stated very simply.

Problem 4 (*Custom instruction covering*): Select a subset C of S that maximizes $M_C(C)$.

In the remainder of this chapter, covering will be analyzed almost exclusively under the further constraint that the chosen subgraphs do not overlap. Before, however, we analyze the components of $M_C(C)$ more closely. This helps understanding the rationale behind the limitation to non-overlapping subgraphs.

The merit function can be chosen more or less arbitrarily. Here we propose a simple merit function estimating the number of clock cycles saved by a particular choice of instruction set extensions. This definition does not depend on the covering algorithm, but rather on the characteristics of the program and the target hardware. Its components are:

- the execution frequency f_i of each graph G_i, to be gathered by profiling,

- a node latency function $\lambda_{sw}(\cdot)$ returning the time needed to execute a node of the graph G when it is not part of a custom instruction (this latency can be fractional if the processor is superscalar);

- a subgraph latency function $\lambda_{hw}(\cdot)$ returning the time needed to execute a subgraph S_{ij} as a custom instruction.

Note that both $\lambda_{sw}(\cdot)$ and $\lambda_{hw}(\cdot)$ are latencies for a single microprocessor instruction; the overall cost of a basic block is the sum of λ_{sw} for all nodes executed as regular instructions, and λ_{hw} for nodes placed in an ISE. Based on this intuition, it is possible to define the merit function in terms of these latencies.

The cost of executing a basic block in software is of course $\sum_{g_i \in G_i} f_i \lambda_{sw}(g_i)$. If parts of it are moved to custom instructions, the λ_{sw} won't have to be included anymore, and on the other hand the λ_{hw} appears for the chosen subgraphs.

It is almost always assumed that the removed nodes are the same that form the chosen subgraphs[1], that is $\bigcup_{S_{ij} \in C} S_{ij}$. Then, the two contributions to the merit function are as follows:

$$\begin{array}{rl} \sum_{g_i \in \bigcup_{S_{ij} \in C} S_{ij}} f_i \lambda_{sw}(g_i) & \text{is the savings from removed instructions} \\ -\sum_{S_{ij} \in C} f_i \lambda_{hw}(S_{ij}) & \text{is the additional cost from hardware execution.} \end{array} \tag{4.1}$$

This however is not enough, as not all values of C may be valid—the constraint that chosen subgraphs should not overlap is an example. Therefore, a predicate $p_C(\cdot)$ is introduced which establishes whether the chosen subset C of custom instructions is acceptable. *This predicate is what really distinguishes different covering algorithms.*

As an example, here are two possible definitions of it. The first corresponds to non-overlapping subraphs, while the second accepts a collection of extensions if it fits the available silicon area A_{max}, with $A_{hw}(S_{ij})$ being the area needed to include a subgraph S_{ij} on the accelerator:

$$p_C(C) \iff S_{ij} \cap S_{ik} = \emptyset \; \forall S_{ij}, S_{ik} \in C \tag{4.2}$$

$$p_C(C) \iff \sum_{S_{ij} \in C} A_{hw}(S_{ij}) < A_{max} \tag{4.3}$$

[1]A notable exception to this definition, as pointed out in Chapter 2, is made by Cong et al. [2004]. The approach used in that paper might occasionally duplicate computations not only across multiple custom instructions, but also between software and hardware. It is possible to fit this in the proposed generic definition of a merit function by introducing a dummy instruction set extensions for each of these duplicated nodes.

These two very different problems have exactly the same formalization except for the validity predicate $p_C(\cdot)$. Note that so far isomorphism has not been taken into account; this would complicate the definition of $p_C(\cdot)$ in equation (4.3), though not in (4.2).

Based on this and on (4.1), the final definition of $M_C(\cdot)$ is as follows:

$$M_C(C) = \begin{cases} \sum_{g_i \in \bigcup_{S_{ij} \in C} S_{ij}} f_i \lambda_{sw}(g_i) - \sum_{S_{ij} \in C} f_i \lambda_{hw}(S_{ij}) & \text{if } p_C(C) \\ -\infty & \text{otherwise} \end{cases} \tag{4.4}$$

Different definitions of $p_C(\cdot)$ may drive the choice of the algorithms used. For example, (4.3) reminds of the definition of the knapsack problem; in fact, a dynamic programming solution can be used successfully in this case [Cong et al., 2004].

In addition, some particular definitions might allow simplifications in equation (4.4), giving rise to more specific problems. This is what happens in the case of non-overlapping subgraphs, given by equation (4.2).

4.2 Non-overlapping subgraphs and generalized exact covers

The main advantage of disallowing overlapping sugraphs is that, because subgraphs of different basic blocks cannot overlap by construction, it is possible to solve the problem one basic block at a time and combine the solutions at the end. This is important even if, as we shall show, the problem remain *NP*-complete within a basic block.

In the context of our framework, however, the constraint of nonoverlapping subgraphs has an additional property that is of great interest. This is the possibility to express $M_C(\cdot)$ in terms of the merit *of single subgraphs* rather than in terms of $\lambda_{sw}(\cdot)$ and $\lambda_{hw}(\cdot)$. This is true because, in this case, each node will only be counted once in the $\sum_{g_i} f_i \lambda_{sw}(g_i)$ term of equation (4.4).

We will call $M(\cdot)$ the merit of a subgraph. When expressed in terms of the node (software) and subgraph (hardware) latencies, its definition is as follows:

$$M(S_{ij}) = -\lambda_{hw}(S_{ij}) + \sum_{g_i \in S_{ij}} \lambda_{sw}(g_i) \tag{4.5}$$

This is exactly the gain computed by the technology mapping passes and, in fact, can be taken as a definition of gain.

When equation (4.5) is used to simplify the definition of $M_C(C)$, this becomes:

$$M_C(C) = \begin{cases} \sum_{S_{ij} \in C} f_i M(S_{ij}) & \text{if no two subgraphs overlap} \\ -\infty & \text{otherwise} \end{cases}$$

While covering a single basic block g_i, it is possible to take out the scaling factor f_i and simply sum $M(S_{ij})$—i.e. the gain from technology mapping—across the chosen subgraphs. The result is the *maximum weighted set packing* problem:

Problem 5 (*Maximum weighted set packing*): Given a binary matrix A, consider all sets of rows such that no column has more than one 1. Given a weight function w_i for each row, find the solution S that maximizes $\sum_{i \in S} w_i$.

This problem is *NP*-complete and is related to the *exact cover* problem, which is also *NP*-complete [Garey and Johnson, 1979]. As in the exact cover problem, in every column there can be at most one 1; however for some columns it may also be acceptable to have *no* 1.

In our case, the matrix has one column per node in the basic block and one row for each candidate subgraph S_{ij}, with ones corresponding to the nodes that are in S_{ij}. Also, all columns may have no 1: this is the same as stating that all subgraphs can be kept as software implementations.

The matrix can be seen as the *incidence matrix* of a hypergraph: the set of nodes in the hypergraph is the same as in the data-flow graph, while each candidate ISE is represented by an hyperedge the includes all the nodes in the ISE. This is shown in Figure 4.1(a).

This problem is solved with a branch-and-bound algorithm by Clark et al. [2006] for a particular merit functions, namely $M(S_{ij}) = |S_{ij}|$. In the general case, it can be reduced to the maximum weighted independent set problem via the following construction, depicted in Figure 4.1(b)(c).

Theorem 4: Let A be the incidence matrix in an instance of Problem 5. It is possible to construct a *conflict* graph with one vertex for each row of A (with weight w_i), and whose edges are created as follows: for each column of A, include a clique consisting of all the rows that have a 1 in that column. The maximum weighted independent set of the conflict graph is also the solution to the maximum weighted generalized exact cover instance.

Proof. Two vertices of the conflict graph are adjacent if and only if there is a column such that both of them have a 1 there; therefore all independent sets are

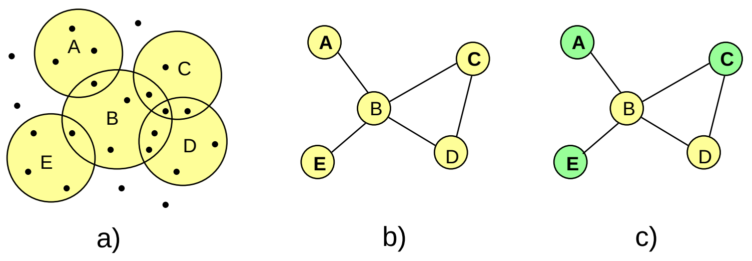


Figure 4.1. Evaluating the merit of a search tree leaf. a) A hypergraph whose hyperedges represent occurrences of custom instructions. b) In the corresponding conflict graph an edge connects two overlapping occurrences. c) An optimal choice of extensions corresponds to a maximum independent set of the conflict graph.

also valid generalized covers and vice versa. In particular, since the weight of an independent set S is also $\sum_{i\in S} w_I$, the maximum weighted independent set is also the maximum weighted generalized exact cover. □

It is interesting to notice the parallel between this construction, which gives the conflict graph in terms of a clique decomposition, and the one of Section 3.2.1, which gave the conflict graph in terms of a bipartite graph decomposition.

This construction is also connected to the concept of a *primal graph* of a hypergraph. The primal graph of a hypergraph H is defined to have the same vertices as H, and edges between all pairs of vertices contained in the same edge of H. Therefore, every hyperedge of H becomes a clique of the primal graph. This leads to the following theorem:

Theorem 5: Let H be the hypergraph whose incidence matrix is A and let H^* be its dual. Then the conflict graph is the primal graph of H^*.

Proof. Since H^*'s nodes are the edges of H, the primal graph of H^* has a node for each edge of H. This is also true of the conflict graph.

H^* has an edge x for each node u of H; x includes each and every edge of H that contains u. Then, each node u of H becomes in the primal graph of H^* a clique, which is formed by all edges of H that include u. Again, this matches exactly the construction of the conflict graph given by Theorem 4. □

In turn, every maximum weighted independent set (MWIS) problem can be easily turned into an instance of problem 5 Let G be the graph for which the

MWIS is sought; then every edge in G can be taken as a 2-hyperedge of H^*. Constructing the dual of H^* gives an equivalent maximum weighted generalized exact cover problem.

4.3 Covering in the presence of isomorphism

So far, we presented covering algorithms that do not cap the number of distinct ISE that are selected. While this can be a viable choice in some cases, in general the target instruction set will pose such a limit. In this case, the ability to detect equivalent ISE and only include them once is very useful in order to enhance the quality of the generated extensions.

Search algorithms that explore the data-flow graph and generate ISE in a single step (for example ITERATIVE, from Pozzi et al., 2006) can perform isomorphism tests during the exploration. For example, the ITERATIVE algorithm exhaustively enumerates all valid subgraphs of a data-flow graph (basic block) and tracks at any time only one subgraph, which is the most profitable subgraph found so far. If it finds a subgraph with the same merit as the current best, ITERATIVE can check the two subgraphs for isomorphism and include both subgraphs in the output.

While this is already quite effective, it has two major limitations. First, since the search works one basic block at a time, this enhancement will not detect replicas that occur in different basic blocks. Secondly, it will always, for example, prefer an ISE which is the biggest but occurs only once, rather than a slightly smaller one occurring twice. The results we'll present in Section 4.4 will show that this case happens quite often in practice.

4.3.1 Establishing isomorphism of candidates

In the framework of Section 2.3, the isomorphism identification pass adds to the covering another input besides the ones of Section 4.1. Isomorphism information consists 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.

Isomorphism detection is treated as a black box. Most past work on the subject looked for structurally isomorphic subgraphs, using generic graph isomorphism algorithms such as *nauty* [McKay, 1981] or *vf2* [Cordella et al., 2004]. In principle, however, the same flow can also accommodate tests for behavioral equivalence. For example, graph rewriting (see also Section 5.2.1) or algebraic techniques [Peymandoust et al., 2003] could be applied to some or all of the subgraphs in the list, and structural isomorphism could be verified on the resulting list.

Since isomorphism detection algorithms are used as oracles, an efficient implementation should not rely only on the speed of the isomorphism test, but it should also try to limit the number of invocations of the test. This is worst-case quadratic in the number of subgraphs, since in principle each of them should be compared with every other subgraph. However, it is possible to partition the subgraphs according to various features (the number of nodes, inputs or outputs; the software cost; the length of the critical path; and so on) and only test isomorphism between subgraphs for which all the features match.

By separating the n subgraphs into p partitions (assumed of equal size), the number of queries will be $O\left(p(n/p)^2\right) = O\left(n^2/p\right)$. The number of partitions is in general high enough that this improvement speeds up the algorithm by several orders of magnitude, and makes an isomorphism-aware compilation flow reasonably fast.

The isomorphism oracle can also be optimized by making sure that the search is pruned effectively. Algorithms such as *nauty* or *vf2* let their clients label the graph vertices and edges arbitrarily, and then prune and speed up the search whenever non-matching labels are encountered.

The meaning of vertex labels is intuitive in a data-flow graph, as it represents either a variable name (for input nodes) or an operator symbol (for expression nodes). Edge labels, instead, identify whether the source node is the LHS or RHS of the operator described by the destination node: the data-flow graphs corresponding to $a - b$ and $b - a$, for example are not isomorphic, because the edges linking a to $-$ are labelled respectively as LHS and RHS. It is possible to implement a very limited kind of behavioral equivalence test by giving the same label to edges leading to a commutative operator, so that the data-flow graphs corresponding to $a + b$ and $b + a$ *are* indeed isomorphic.

4.3.2 Greedy covering

The challenge is now to devise a covering algorithm that processes subgraphs labeled with merit information $M(\cdot)$ and with isomorphism information.

The one presented in Figure 4.2 proceeds greedily, picking the best candidate ISE and its occurrences (a set of non-overlapping isomorphic subgraphs) at every step. The algorithm alternates between two phases, namely finding a valid covering for a single candidate, and determining if that candidate can be the next greedy choice.

```
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 $N \cap S = \emptyset \quad \wedge$
             there are no $o_1, o_2 \in O, s_1, s_2 \in S$ such that
             there is a path from $o_1$ to $s_1$
             and a path from $s_2$ to $o_2$ then
                $N = N \cup S$
                $O = O \cup$ outputs of $S$
                $OCC = OCC \cup \{S\}$
    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($H, M(c), c$)
    while HEAP-MAX($H$) $\notin V$ do
          $c =$ HEAP-EXTRACT-MAX($H$)
          $OCC =$ COVER-SINGLE-CANDIDATE($c, chosen$)
          HEAP-INSERT-NODE($H, M(OCC), OCC$)
          $V = V \cup \{OCC\}$
    return HEAP-MAX($H$)
```

Figure 4.2. A greedy covering algorithm.

The first phase (COVER-SINGLE-CANDIDATE) eliminates overlapping replicas from each candidate set of isomorphic subgraphs. The reason for this step is that of course the covering algorithm may only choose one among all the subgraphs that include a given program node. In particular, when a node appears in several isomorphic subgraphs, this phase associates each node to at most one such subgraph. After choosing the first ISE, in addition, this phase will also discard subgraphs that overlap a previously chosen extension.

This selection process is run on the family of isomorphic subgraphs that has the highest merit. What is left to do for a greedy algorithm is then to recognize whether this family is in fact the single best possible ISE. If this is not the case, the algorithm shall find another candidate for which to find a valid covering.

A simple and efficient implementation will be described in the remainder of this section. We call a candidate *valid* if its covering has already been computed by COVER-SINGLE-CANDIDATE; otherwise the candidate is termed *invalid*. The overall merit of a candidate is the sum of all the merits from each occurrence; and since COVER-SINGLE-CANDIDATE will only discard some copies of the subgraph, it may only decrease the merit of the candidate. Therefore, the merit of an invalid candidate may overestimate the real figure, while the merit of a valid candidate is correct.

The algorithm then looks repeatedly at the candidate with the highest maximum merit. If it is valid, it can be chosen immediately; otherwise the merit may be overestimated, and we need to find the candidate's cover using COVER-SINGLE-CANDIDATE—after which the candidate is valid, and the algorithm repeats. After each call to COVER-SINGLE-CANDIDATE, the algorithm may or may not be chosen depending on how much the valid covering caused its merit to decrease.

The efficient implementation of the algorithm is based on a binary heap data structure. Heaps store a set of values, each associated with a key, and allow a fast ($O(\log n)$) implementation of two operations: *insert-node* (insert a key-value pair), *extract-max* (delete the pair whose key is maximum[2]). The maximum can also be read non-destructively (operation *max*) in constant time. The algorithm using this data structure is also shown in Figure 4.2.

Multi-step greedy covering. In the framework of Section 2.3 the covering phase may rely on more than one technology mapping algorithm. By evaluating the hardware cost of a given subgraph with increasing detail—theoretically, down to the level of logic synthesis—it is possible to restrict more expensive analyses only to the best candidates, and dually, to run only the fastest technology mapping routines on the entire list of subgraphs.

The algorithm easily extends to the case when multiple technology mapping subroutines are available, As remarked in Figure 4.3, in this case the merits

[2]Of course, heaps can be implemented so as to extract the pair whose key is minimum. In general, however, only one of *extract max* and *extract min* can be implemented, short of coupling two effectively independent data structures.

```
COVER-MULTI-TM($candidates, chosen, maxtm$)
    ▷ $candidates$ 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
    ▷ For every candidate $c$, merit functions obey $M_1(c) > M_2(c) >$
        $> \ldots > M_{maxtm}(c)$
    $H$ = HEAP-NEW
    for each element $c$ in $candidates$ do
        HEAP-INSERT-NODE($H, M_1(c), c, 0$)
    while true do
        $(c, pass)$ = HEAP-EXTRACT-MAX($H$)
        if $pass = maxtm$ then
            return $c$
        if $pass = 0$ then
            $OCC$ = COVER-SINGLE-CANDIDATE($c, chosen$)
        $pass = pass + 1$
        HEAP-INSERT-NODE($H, M_{pass}(OCC), pass$)
```

Figure 4.3. Supporting multiple technology mapping steps.

computed by the subroutines on a subgraph S will be $M_{C1}(S) > M_{C2}(S) > \ldots > M_{C\,maxtm}(S)$, and the algorithm will refine the estimate for the top element of the heap until it reaches the most precise one.

4.3.3 Optimal covering

Clark et al. [2006] give an optimal branch-and-bound covering algorithm for the special merit function $M(S_{ij}) = |S_{ij}|$. 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. In this case the merit of a candidate of size s is $s - 1$. Adding 1 to the merit 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.

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 [Pozzi and Ienne, 2005; Verma et al., 2007];

- 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 [Biswas et al., 2006]. This is the case for the experimental platform we used, since it allows the accelerator to include read-only memories.

Therefore, we present a different optimal solution to the covering problem (with isomorphism, non overlapping solutions, and a bounded number of instructions U_{max}).

Because of the bound on the number of chosen instruction, a first idea could be to perform exhaustive search on the whole universe of subgraphs S; given $|S|$ candidate ISEs, the worst-case complexity of this approach would be $O\left(|S|^{U_{max}}\right)$. Of course, since $|S|$ can be as high as 10^6, this has to be augmented with pruning strategies to be usable in practice.

Optimizing exhaustive search. First of all, we will describe a branch-and-bound criterion for this algorithm, described in Figure 4.4(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 merit that they can contribute (from highest to lowest), and higher-merit classes are always tried first.

Let $m_1, ..., m_n$ be the merit of each equivalence class. Then, at any point, if C is a partial solution, m is its merit, and the next equivalence class to be examined is U_k, the merit bound is $m_{max} = m + \sum_{k \le i < k+U_{max}-|C|} m_i$. If the best merit achieved so far is greater than or equal to m_{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 merit of the entire solution (Problem 5). A secondary pruning criterion, shown in Figure 4.4(b), can be used to avoid evaluating the merit of each and every leaf of the search tree. To this end, for each subgraph in the solution (including

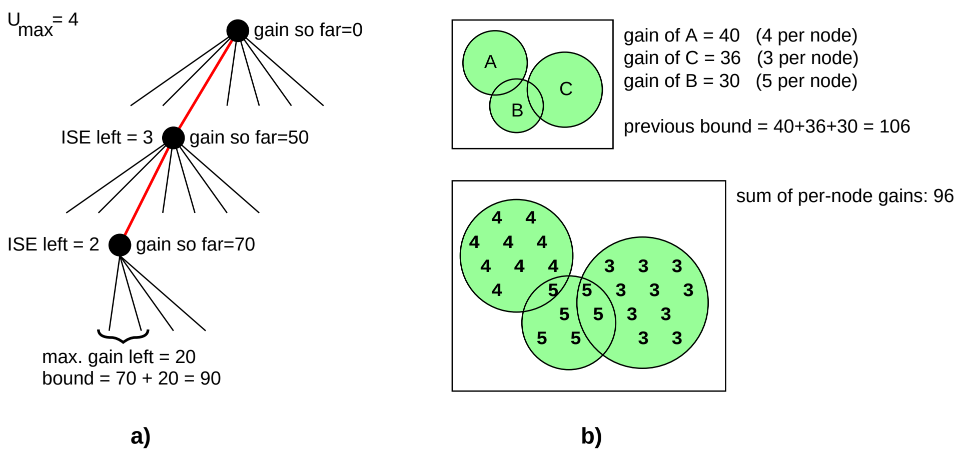

U_{max} = 4
gain so far=0
ISE left = 3
gain so far=50
ISE left = 2
gain so far=70
max. gain left = 20
bound = 70 + 20 = 90
a)
A
B
C
gain of A = 40 (4 per node)
gain of C = 36 (3 per node)
gain of B = 30 (5 per node)
previous bound = 40+36+30 = 106
sum of per-node gains: 96
b)

Figure 4.4. 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 merit, plus the merit of the next n ISEs to be examined. b) For search tree leaves, the bound is refined to skip the evaluation of MWIS 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 merits" criterion only counts the three overlapped nodes once.

---

overlapping copies) we compute a *per-node merit*:

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

Then, we compute the maximum merit 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 per-node merit. The merits of all the nodes are then summed, yielding an upper bound to the solution's merit[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 merit, and therefore it decreases the number of MWIS instances to be solved.

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

```
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($\{s \in S : \exists s' \in C_{prev},\ s \cap s' \neq \emptyset\}, count$)
7     while $C \neq C_{prev}$
8     $C_{greedy} = $ GREEDY($\{s \in S : \forall s' \in C,\ s \cap s' = \emptyset\} \cup C, count$)
9 while $C_{greedy}$ better than $C$
```

Figure 4.5. An iterative, optimal covering algorithm

Despite these improvements, however, exhaustive search is still impractically expensive and, for all but the smallest inputs, even a solution with $U_{max} = 2$ may require unreasonable time. For this reason, we developed a more practical optimal algorithm, which couples exhaustive search (on a reduced universe) with greedy search.

Optimal hybrid search. In this algorithm, called *hybrid search*, exhaustive search is used to improving a pre-existing solution, while greedy search ensures that the search does not get stuck on a local optimum (which would be possible because of the reduced universe).

This algorithm is based on the following observation:

Theorem 6: 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 $m_1 \geq m_2 \geq \ldots \geq m_{U_{max}}$ are the total merits 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-merit set of isomorphic subgraphs in C^+, and let its merit be m_k^+; k is the lowest value such that $m_k^+ > m_i\ \forall i : k \leq i \leq U_{max}$. Such a set exists because the total merit m^+ of C^+ is higher than the total merit of C (if it does not exist, $m^+ < U_{max} m_{U_max} < m$ leading to a contradiction). Since UC_k^+

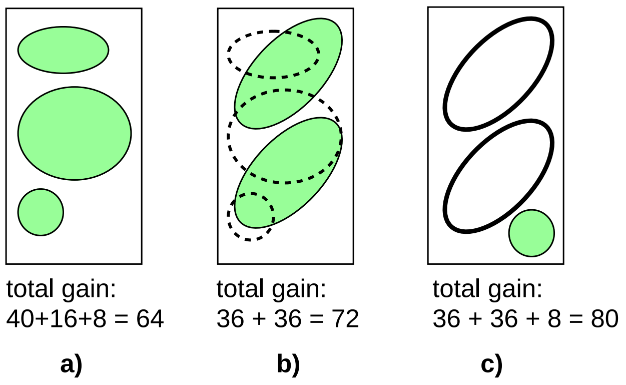


Figure 4.6. Sample execution of the optimal hybrid algorithm. (a) Result of greedy search. (b) Exact search is run on a reduced universe to improve the previous solution. (c) Greedy search is run again to find solutions not explored by the previous steps.

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 4.5. 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 4.6. Figure 4.6(a) shows the solution S_1 found by the greedy algorithm, consisting of three ISEs that, together, achieve a merit of 64. Figure 4.6(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 merit. A further greedy run adds a third ISE to the current covering and obtains solution S_3, shown in Figure 4.6(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 7: The algorithm terminates.

```
HYBRID($S, count, depth$)
 1  $C = \emptyset$
 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          $C = C \cup C_{step}$
 7      else
 8          $UC_1$ = most profitable group of isomorphic subgraphs in $C_{step}$
 9          $C = C \cup UC_1$
10          $S = \{s \in S : \forall s' \in UC_1,\ s \in s' = \emptyset\}$
```

Figure 4.7. An approximate covering algorithm

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 merit or reduce the number of distinct ISEs (which cannot happen U_{max} or more times). Since the maximum merit achievable is bounded, the algorithm will converge. □

Theorem 8: 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 $m_1, ..., m_{U_{max}}$ be the total merits 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 merit m_k^+; k is the lowest value such that $m_k^+ > m_i\ \forall i : k \leq i \leq U_{max}$. Such a set exists because the total merit m^+ of C^+ is higher than the total merit of C (again, if it does not exist, $m^+ < U_{max} m_{U_max} < m$ leading to a contradiction).

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 merit) 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. □

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

4.4 Experimental results

The experimental results presented in this section demonstrate the effectiveness of isomorphism-aware covering, and compare the speedup achievable by the algorithms of Figure 4.2 (greedy) and 4.7 (d-optimal).

We implemented these techniques in a toolchain for extensible processors based on GCC, extended as described in Chapter 6, and SimpleScalar/ARM. The reader should refer to Section 6.3 for the details of the simulated architecture.

The implementation follows the framework of Section 2.3, using exact algorithms for enumeration and isomorphism check. For enumeration we used the I/O constrained algorithm of Section 3.3; for isomorphism we used the *vf2* algorithm [Cordella et al., 2004], a bottom-up algorithm that is well suited to graphs that exhibit regularity, and only needs space 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 [Minato, 1993]. These allow a memory efficient representation of combination sets and support complex operations on them, including extraction of overlapping sets from a universe [Okuno et al., 1998].

For comparison with the state-of-the-art, we also implemented an alternative approach that is able to map multiple equivalent ISE onto the same instruction, but on the other hand does not use isomorphism to guide the search. This

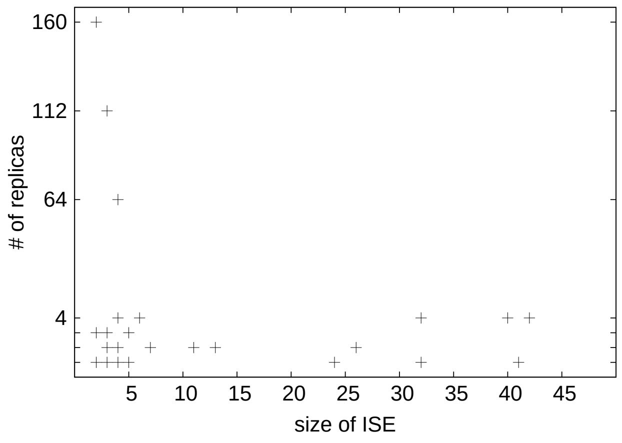


Figure 4.8. Extensions obtained from the seven benchmarks we used, plotting the instruction's size versus the number of replicas found. (The top half of the y axis uses a different scale).

algorithm simply picks the highest-merit candidate in a basic block repeatedly, until it finds $U_{max} + 1$ distinct extensions; then it keeps the U_{max} best choices[4].

The algorithms are evaluated on seven programs from the MiBench suite [Guthaus et al., 2001].

First of all, we analyze the effectiveness of our proposed algorithms, as well as of recurrence-aware ISE search. Figure 4.8 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 greedy covering algorithm with a constraint of 4 inputs, 2 outputs.

Two notable facts can be observed. 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 mostly occur in the aes benchmark, for which a non-recurrence-aware algorithm cannot find any profitable extension. Instead, 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, recurrence-aware covering is *not* limited to finding small extensions that, like multiply-accumulate instructions, many processors already implement.

[4]The lookahead is needed to find replicas of the U_{max}-th best choice.

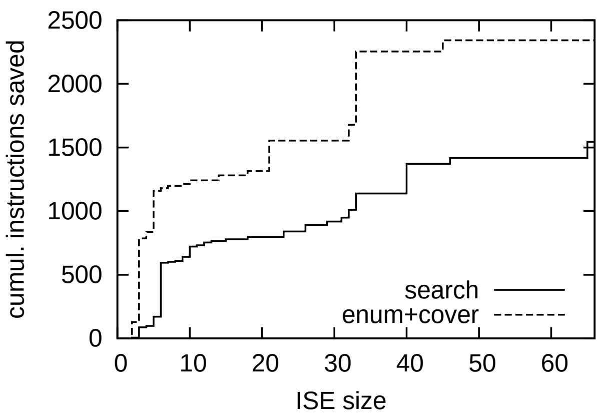


Figure 4.9. Cumulative number of instructions that were moved to hardware, as the size of the ISE varies. Overall, state-of-the-art algorithm saved 1,500 (static) instructions in the seven selected benchmarks; our technique saved 2,400.

Another view of the same data is offered by the cumulative histogram in Figure 4.9. This chart plots how many instructions are removed (Y axis) by instruction set extensions up to a certain size (X axis). Every step on the Y axis indicates an ISE size (on the X coordinate) that was selected in the benchmarks. Steep increases correspond to ISE sizes that save a high number of (static) instructions.

The top line refers to the method of Figure 4.2, while the bottom line represents the current state of the art. Increases are more pronounced for the top line, meaning that the algorithm we presented is more effective than the state of the art; they are also shifted to the left, showing that the new implementation may sometimes prefer smaller instructions (20-30 nodes) to bigger ones (40-60 nodes) that don't have any replica.

The very steep increase at the extreme left of the chart also corresponds to aes, where only small extensions (< 10 nodes) can be found and the new algorithm is much more effective.

Figure 4.10 shows the speedups obtained with greedy covering for different I/O constraints and algorihm. Note that the speedups obtained are often very tangible. The inherent problems of employing a greedy algorithm are also evident here from two phenomena: first, the isomorphism-aware algorithm is in some cases (sha, rawcaudio) slower than the state-of-the-art; second, the speedup of the 4-2 constraint sometimes is inferior to that of lower constraints.

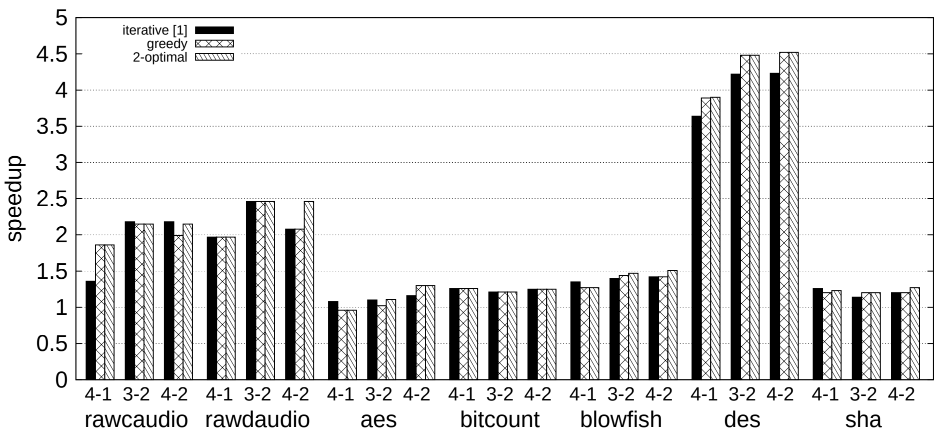


Figure 4.10. Speedups obtained with different covering algorithms and I/O constraints.

	Iterative	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 4.1. 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.

The latter kind of failure, detailed in Table 4.1, can complicate design space exploration, since switching to higher constraints will not necessarily improve performance. Most of these effects can be avoided by switching to the d-optimal algorithm; 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. Residual anomalies (e.g. aes and blowfish, 4-1) are due to the difference between the estimates that the search algorithms work on, and the actual gains obtained after simulation.

Another point to analyze is the effectiveness of the pruning heuristics. The results are in Table 4.2. For each I/O constraint, the first column gives the ratio

	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 4.2. Ratio of considered candidates to the total number of enumerated candidates, and number of search graph leaves that branch-and-bound cannot exclude.

	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 4.3. Compilation time for the greedy algorithm, as well as for increasingly accurate approximate searches (4 inputs, 2 outputs).

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 MWIS 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 4.3 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.

Chapter 5

Technology mapping

This chapter analyzes a sample technology mapping methodology for custom instructions, thus concluding the presentation of the proposed compilation framework. In particular, we consider an example of coarse-grained reconfigurable architecture, the EGRA (Expression-Grained Reconfigurable Array). Coarse-grained reconfigurability is often proposed as the solution to the flexibility problems of ASIC and the performance problems of fine-grained platforms such as FPGAs. However, compiling for CGRAs poses peculiar problems because of the unique combination of spatial execution abilities with arithmetic-level programmability.

In this chapter, we first introduce the EGRA and its main processing element—the *Reconfigurable ALU Cluster* (RAC). We then propose a mapping flow for EGRAs that estimates the advantage of executing the subgraphs on hardware and at the same time produces the program bitstream for the EGRA. The chapter is concluded with the presentation of a theoretical result, namely, an NP-completeness proof of *I/O scheduling*, an important step of a compilation flow for customizable processors [Pozzi and Ienne, 2005; Verma et al., 2007].

5.1 The EGRA architecture

In the earliest examples of reconfigurable architecture such as the PLA (Programmable Logic Array), mapping of "applications" (Boolean formulas in sum-of-product form) is immediate. As in Figure 5.1(a), each gate in the application is mapped in a *1-to-1* fashion onto a single gate of the architecture).

However, this organization does not scale as applications to be mapped get more complex. For this reason, CPLDs and FPGAs instead use elementary components—PLAs themselves, or look up tables as in Figure 5.1(b)—as build-

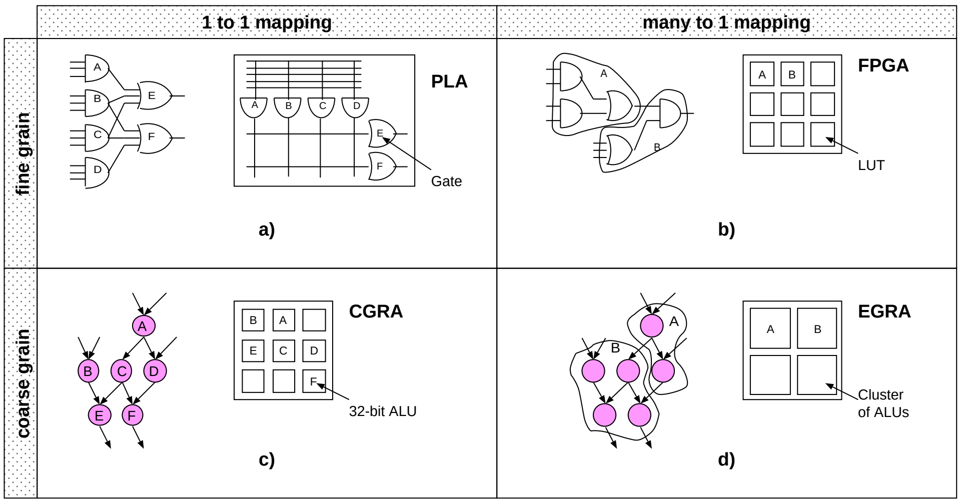


Figure 5.1. Parallel between the evolution of fine-grained architectures from simple programmable devices to FPGAs (a and b), and the evolution of CGRAs from simple cells to the EGRA proposed here (c and d).

ing blocks, and glue them with a flexible interconnection network. Then, programming one cell corresponds to identifying *more than one gate* in the Boolean function representation.

Introducing this additional level is a winning architectural choice in terms of both area and delay, but such innovations cannot be successful unless algorithms are available to efficiently map applications to the new architecture—and indeed efficient algorithms came along to this purpose, e.g., FlowMap [Cong and Ding, 1994].

An orthogonal step was the introduction of higher granularity cells—see Figure 5.1(c). Fine-grained architectures provide high flexibility, but also high inefficiency if input applications can be expressed at a level coarser than boolean (e.g. as 32-bit arithmetic operations). Coarse-Grained Reconfigurable Arrays (CGRAs) provide larger elementary blocks that can implement such applications more efficiently, without undergoing gate-level mapping.

A variety of CGRA architectures exist (see Section 2.1) but the process of mapping applications to current CGRAs is usually not very sophisticated: a single node in the application intermediate representation gets mapped onto a single cell in the array (again, *1-to-1* mapping). Instead, we propose to employ an array cell that consists of a group of programmable ALUs, supporting efficient computation of entire subexpressions in a single cycle. This cell is the RAC, and

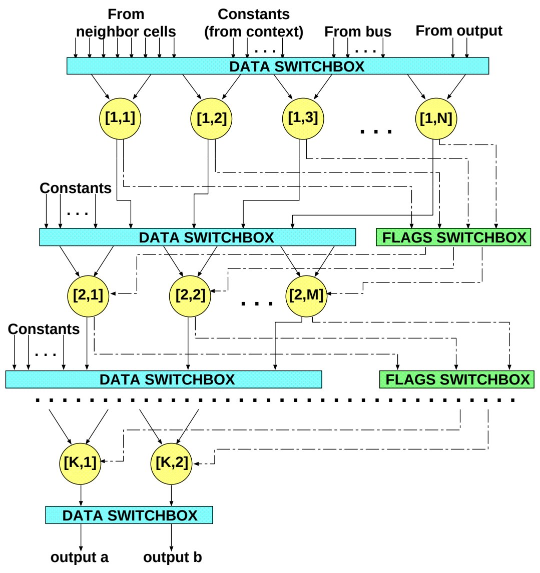


Figure 5.2. Datapath of a Reconfigurable ALU Cluster (RAC).

the resulting array architecture (Figure 5.1(d)) is the EGRA; we consider the introduction of the RAC as the moral equivalent of the switch from single gates to LUTs that characterizes modern fine-grained reconfigurable architectures.

5.1.1 Cell architecture

The RAC datapath consists of a multiplicity of ALUs, with possibly heterogeneous arithmetic and logic capabilities, and can support efficient computation of entire subexpressions, as opposed to single operations. It is inspired by the Configurable Computation Accelerator proposed by Clark et al. [2004]. However, we use this structure as a *replicable* element; this has important consequences. First of all, it opens up the possibility to create combinational structures using multiple RACs; unlike Clark's design, in which a CCA has a fixed multi-cycle latency, this favours designs featuring a smaller number of rows and with a latency smaller than half a cycle. Furthermore, it removes the limit on the number of inputs and outputs, because a pipelining scheme [Pozzi and Ienne, 2005] can be used to move data in and out of the array; this allows scheduling of more complex applications and consequently higher gains.

ALUs are organized into rows (see Figure 5.2) connected by switchboxes. It is important to have flexible routing between ALUs on neighbouring rows, be-

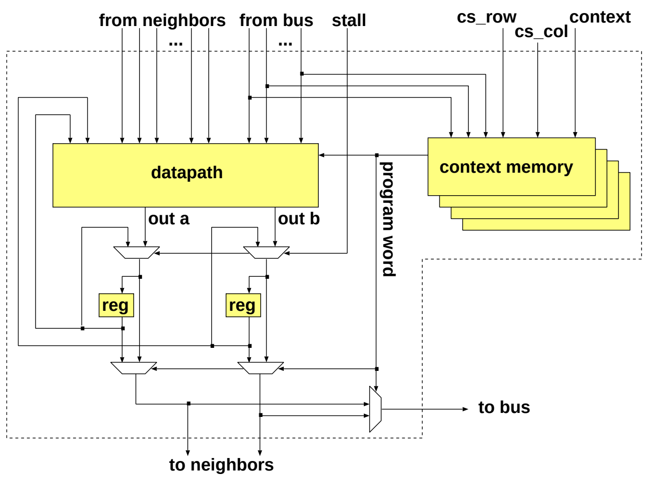


Figure 5.3. Block scheme of a RAC composed of datapath, context memory and bypassable registers on the outputs.

cause subexpressions extracted from typical embedded applications often have complex connections that are not captured well by simpler topologies. This organization allows the usage of a simple array topology (we used nearest neighbour) without incurring high penalties on place-and-route.

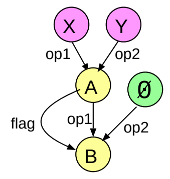
The inputs of the RAC (see again Figure 5.2) are taken from the neighbouring cells' outputs, from the outputs of the cell itself, or from a set of constants specified separately for each RAC; the inputs of the ALUs in subsequent rows are routed from the outputs of the previous rows or again from the constants.

The number of rows, the number of ALUs in each row and the functionality of the ALUs is flexible and can be customized for different applications, for example depending on cycle time and area constraints.

The number and size of the constants is also defined at exploration time. If their width is less than the datapath width, their content is zero-extended[1]. The value of the constants, instead, is part of the configuration bitstream and can be different for each cell.

ALU design. As in other CGRAs, the basic processing element of our cell design is an ALU. Unlike in the fine-grained domain, it is not possible to define a generic component that can implement arbitrary functions, as is the case of the PLD or

[1]The availability of operations such as $A+\overline{B}$ makes it possible to store negative values even if the constants themselves are zero-extended.



node	opcode	op1 source	op2 source	flag source
A	op1 + $\overline{\text{op2}}$ + flag	BUS input 1	BUS input 2	1
B	flag ? op1 : op2	dataout(A)	constant 0	GEU(A)

Figure 5.4. Programming a RAC. This example shows how two ALUs can be connected to compute an unsigned subtract with saturation, `(X >= Y) ? X - Y : 0`. The node computing the subtraction also performs the comparison. The multiplexer node B uses both the data output and the *unsigned* $\geq$ flag of the subtraction node A.

data opcodes	flag opcodes
$out = A \,\&\, (B \oplus flag_{sext})$	0
$out = A \mid (B \oplus flag_{sext})$	1
$out = A \oplus (B \oplus flag_{sext})$	$=$
$out = flag \;?\; A \;:\; B$	$\neq$
$out = A + B + flag_{zext}$	signed $<$
$out = A + \overline{B} + flag_{zext}$	signed $\geq$
$out = A \ll B$	unsigned $<$
$out = A \ll_{rot} B$	unsigned $\geq$
$out = A \gg_{arith} B$	
$out = A \gg_{logical} B$	
$out = A \gg_{rot} B$	

Table 5.1. List of supported opcodes. Note that the 1-bit flag input will be sign- or zero-extended depending on the opcode.

the LUT. Therefore, expressions are realized in our architecture by clustering more than one elementary unit (ALU) in one cell.

Four types of ALUs can be instantiated. The simplest one is able to perform bitwise logic operations only; the other three add respectively a barrel shifter (with support for shifts and rotates), an adder/subtractor, and both the shifter and adder. The list of operations in a full-featured unit is in Table 5.1.

A peculiar element in the design of the RAC is *flags*. These are inspired by the program status word of a microprocessor, and are significantly more powerful than the carry chains available in many reconfigurable architectures (for example, Stretch [Rupp, 2003] or PipeRench [Goldstein et al., 1999]). They enable efficient implementation of if-conversion, which is important when automatically mapping software representations onto hardware. ALUs can act as a multiplexer, choosing one of the two 32-bit inputs based on another ALU's flags; then, one or more cells will evaluate both arms of a conditional, and a multiplexer will choose between the two.

The operation of flags is explained graphically in Figure 5.4. First, each operation can generate three 1-bit flags: a *zero flag* used for equality comparisons, an *unsigned* $\geq$ *flag* (equivalent to the carry flag of general-purpose processors), and a *signed* $<$ *flag* (equivalent to $N \oplus V$, where N and V are the sign and overflow flags). Dually, each operation has *three* operands, two being 32-bit values and the third being a 1-bit value. The third operand can be hardcoded to zero or one, or it can come from the flags that another ALU generated; incoming flag values can also be complemented, thus giving a total of eight possible *flag opcodes* (also listed in Table 5.1). Complementing the flag, and/or exchanging the operands of the comparison, allows to test all possible conditions on both signed and unsigned operands.

5.1.2 Array architecture

The EGRA architecture is composed of a collection of RACs with the described datapath (cells). Each of them also has a *context memory*, included in Figure 5.3, which stores a number of configuration words allowing the cell to be programmed according to the desired functionality.

The array is connected to the processor bus: inputs can be taken from the register file and broadcast to the cells, while outputs can be sent to the register file from selected cells per cycle (one per write port in the register file).

In addition to the cells, the EGRA includes a global control unit. This unit is in charge of managing the transfers to the RACs' context memory, selecting contexts, stalling cells until their output data is consumed, and connecting their outputs to the bus. The control unit is also programmable through its own context memory. Since they are not relevant to the implementation of technology mapping, we do not detail the structure of the control unit, the connection between the cells, and the possible extensions to the model. We urge the reader instead to refer to existing publications on the subject; Ansaloni et al. [2008] describe the architecture of the EGRA, while Ansaloni et al. [2009] analyze the advantages of introducing heterogeneous processing elements, such as local memories and specialized arithmetic components (in particular multipliers).

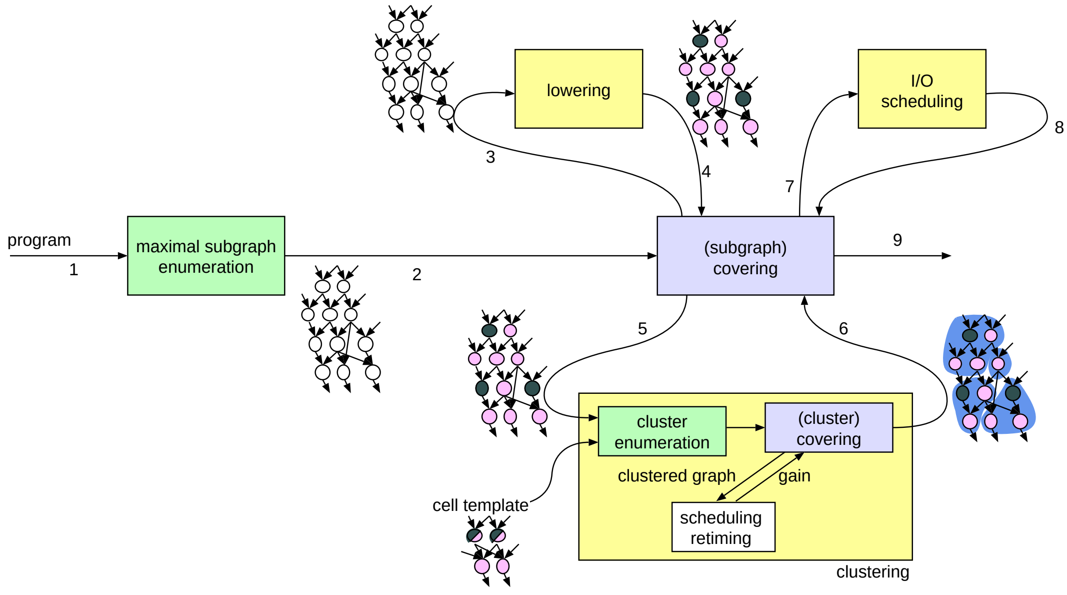


Figure 5.5. Scheme of the technology mapping passes for the RAC compilation flow.

5.2 Technology mapping for the EGRA

The proposed mapping flow, which is designed to fit the framework discussed in the previous chapters, is drawn in Figure 5.5.

The proposed mapping flow is designed to have as input maximal subgraphs, extracted through the algorithm of Section 3.2.2. Each maximal subgraph is called a candidate; in order to measure its gain, it needs to be undergo several additional steps that are grouped in three technology mapping passes: *lowering*, *clustering*, *I/O scheduling*.

The former converts the data-flow graph to a form similar to that of Figure 5.4, where all operations are expressed through the opcode set of the RAC. For example, C logical operations are rewritten to use the RAC's flags; lowering is represented in Figure 5.5 by filled circles in the candidate's data-flow graph. This step is detailed in Section 5.2.1.

Clustering instead is more complex and can be decomposed further in several steps: cluster enumeration, cluster covering and scheduling/retiming. These three steps are in fact a "mini-occurrence" of the same mapping framework, which is used in this case to map a single ISE onto many RACs. After this step, the bitstream for the EGRA is known and each operation is assigned to a particular ALU of a particular RAC. The enumeration and covering techniques are described in Section 5.2.2, while the rationale and techniques for scheduling and retiming (the “inner” technology mapping steps) are described in Sections 5.2.3 and 5.2.4.

The last technology mapping step is I/O scheduling. Since the subgraphs do not need to have any constraint in term of number of inputs and outputs; in case these exceed the bandwidth of the register file, this pass serializes them across multiple cycles. For this step we do not describe any novel algorithms, referring instead to those of Pozzi and Ienne [2005] and Verma et al. [2007]; as anticipated, however, Section 5.3 proves the *NP*-completeness of this problem.

More complex tasks are in later technology mapping passes by design. As mentioned in Section 4.3.2, not all of the candidate maximal subgraphs go through all of this steps. The lowering step for example, while being fast to execute, will already compute simple lower bounds on the cycles needed to execute the candidate on the accelerator (and equivalently, upper bounds on the gain). These bounds can also conservatively include the register file bandwidth and the number of inputs and outputs, even though the actual scheduling of register file accesses occurs only at the end. This way, the framework avoids executing the expensive steps for clearly suboptimal subgraphs.

5.2.1 Lowering to the RAC instruction set

The first technology mapping step, operation lowering, enables effective usage of the capabilities of the RAC, in particular the flags architecture. Flags can be used to drive multiplexers, thus implementing if-conversion as in the example of Figure 5.4, but they will also be used to implement C logical operations (i.e. using the truth value of a comparison in an arithmetic operation) and to transform special nodes that are included in the intermediate representation, such as absolute value and minimum/maximum.

In all cases, comparisons must be transformed to an operation (a subtraction or an exclusive OR) that will compute flags; users of the comparison can then test a particular condition on those flags. In addition, operands of the comparison can be manipulated in order to support conditions that the ALUs cannot generate, i.e. $>$ and $\leq$.

These transformations are implemented using graph rewriting; this was a natural choice since the entire compilation flow is based on a data-flow graph intermediate representation. In this system, the set of lowering rules constitutes a grammar, with each rule is described by a pair of graphs—a *template* to be matched in the candidate data-flow graph, and a *replacement* graph that will

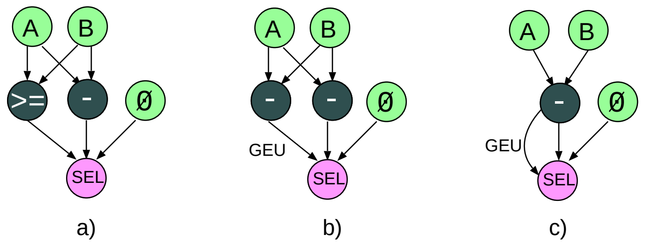


Figure 5.6. Lowering a data-flow graph to the set of operations supported by the cells. This example shows a saturating unsigned subtraction. a) The DFG representing the C source `A >= B ? A - B : 0`. b) The comparison is transformed into a subtraction and the appropriate flag (unsigned greater-than-or-equal) is tested in the SEL node. c) Common subexpression elimination merges the two subtraction nodes into one.

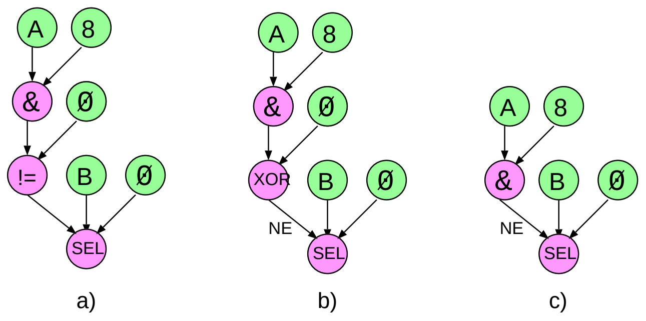


Figure 5.7. Peephole optimization during lowering. a) Example is taken from the ADPCM benchmark. b) Equality and inequality comparisons are converted to exclusive-ORs. c) Comparisons with zero do not need a separate node, because the AND node is already computing the right value of the flags.

be substituted for it. Graph rewriting is a very powerful technique, and several libraries are available to program graph rewriting systems. In particular, we used the GrGen library [Geiß et al., 2006].

The graph grammar for lowering is small, consisting of only 10 rules; example applications of these rules are shown in Figures 5.6 and 5.7. In both cases, the leftmost graph in the figure shows the intermediate representation coming out of compilation, including comparison operators such as $\geq$ and $\neq$. These are transformed into flag tests in the second graph; for equality/inquality tests they can be transformed into an exclusive OR, while other tests are changed into a subtraction.

Figure 5.6(c) and 5.7(c) show that additional simplifications can then be performed, in particular CSE and peephole optimization. These steps were also described as graph rewriting rules, using approximately the same number of rules as for lowering. The common subexpression elimination optimization is shown in Figure 5.6(c), and it allows to use both the numeric and flag outputs of a single node. The peephole optimization example of Figure 5.7(c) instead shows how comparisons with zero can be merged in the node that computes the other operand of the comparison. In both cases, this results in less strict requirements for the number of ALUs, and possibly in a shorter latency for the instruction set extension.

5.2.2 Partitioning of the candidate

The lowering step produces a new DFG, where the set of used operations matches closely the capabilities of a single ALU of the EGRA. In other words, technology mapping defines how the array's ALU will execute each operation in the data-flow graph. The next step is to coarsen the granularity, and find a mapping at the RAC level; to this end, the partitioning phase clusters operations that will execute in the same cell. The output is a new graph that collapses these clusters to a single node; this graph more closely represents the execution of the candidate DFG on the EGRA.

Covering the candidate with a number of clusters is in turn a multi-step process, comprising enumeration of the possible clusters, analyzing their timing characteristics (for example eliminating those that exceed the chosen cycle time), and then selecting, among the enumerated and valid ones, a set of clusters that partitions the candidate.

Clusters can also be seen as data-flow graphs. The most important difference between the candidate subgraphs enumerated as the first step of the process, and the cluster graphs considered here, is that clusters are constrained to the set of computations that a cell can perform. The constraints are as follows:

1. clusters should be convex;
2. clusters should have no more outputs than the RAC has;
3. the maximum depth of the cluster is constrained by the number of rows in the RAC;

4. not all rows of the RAC can execute all operations;

5. cluster rows should not be wider than RAC rows.

The third constraint is important, as it allows adaptation of the algorithms from Section 3.3, in particular the data-flow based technique of Figure 3.15. While the RAC does not have a limit on inputs, the constraints on depth and on the kind of operation can be performed limit the number of clusters that are enumerated.

The last example of Section 3.3 already showed the definition of a simple lattice implementing depth-constrained enumeration. We can improve on that result so that the fourth constraint above is also checked at enumeration time.

Since the generic data-flow-based algorithm takes care of convexity and of the number of outputs, enumeration successfully filters the first four constraints above. The width of the RAC rows is checked instead after the operations in the cluster are assigned to the ALUs in the RAC (*scheduling*; see Section 5.2.3).

The value that we propagate along the nodes of a candidate is the *set of rows of the cell* that can implement the node's operation. This is efficiently stored in a bitmask, using a bit per cell row. This bitmask has two components, one that depends on the data-flow analysis and one that is static.

The latter is computed beforehand for every opcode used by the intermediate representation, and its value can be used to initialize the field when a node is added to the cluster; we will call this per-opcode bitmask the *op-mask*. So, for the example RAC of Figure 5.8(a), the op-mask of *SEL* nodes[2] is *110*. In this case, the value *110* includes rows that support logic operations (all three do) and have a flag input (the two bottom rows). The op-mask for addition is also *110* (this time, because the top row does not have an adder) and the op-mask for shifts is *101*.

The second component instead is computed as follows. If b_j is the bitmask of node j, the index $MSB(b_j)$ of the most significant set bit of b_j indicates the last RAC row in which b_j can run; predecessors of j will have to be allocated on a row in the range 0 to $MSB(b_j) - 1$. The data-flow–sensitive bitmask is then $(1 \ll MSB(b_j)) - 1$, and this value is ANDed with the op-mask to obtain the bitmask for the newly added node.

This corresponds to the following definitions:

- given a bitmask m (the *OUT* bitmask for a node), $f(m)$ (the corresponding *IN* bitmask) is equal to $(1 \ll MSB(m)) - 1$.

[2]Since masks are read like binary numbers, the most significant bit (corresponding to the bottom line of the RAC) is written first.

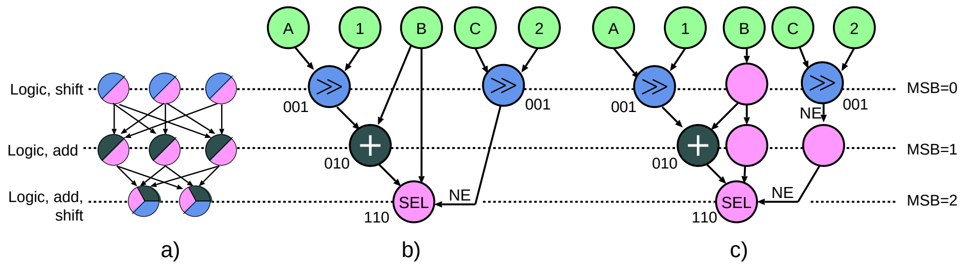


Figure 5.8. Scheduling operations in a RAC. a) A cell template. b) A cluster that has edges spanning multiple rows. The enumeration algorithm assigns bitmasks to each operation, corresponding to the rows on which the operation can be executed. c) By placing ops on the row indexed by the MSB, one obtains the as-late-as-possible schedule; in this case, 2 ALUs must be configured to pass the value through, and 1 must pass the equality flag.

- the per-node data-flow function *limit*(v) is equal to v's op-mask.
- given two bitmasks a and b, $a \wedge b$ is equal to the logical AND of the bitmasks.
- $\perp$, the lattice value that causes the subgraph to be declared invalid and the search to be pruned, is the empty bitmask 0.

After a list of clusters is found, finding a partitioning is equivalent to *covering* the candidate with clusters—as mentioned earlier, partitioning is in fact another occurrence of the same mapping framework analyzed throughout this thesis. It is possible to use a greedy covering algorithm that tries to place the deepest available cluster on the critical path. Deep clusters have higher utilization rates, and minimize the routing delay on the critical path both within a RAC and in the entire array.

5.2.3 Cluster scheduling

The purpose of scheduling is to ascertain the validity of the cluster and inserting pass-through operations for values and/or flags, whenever they are needed. This process is closely related to retiming, because the way operations are scheduled influences the critical path delay of the RAC, and consequently the latency of the computation on the EGRA.

The starting point for scheduling are the bitmasks obtained by the dataflow enumeration algorithm, shown in Figure 5.8(b). The simplest possible scheduling algorithm uses the MSB of the bitmask to allocate each node to a RAC row; this corresponds to as-late-as-possible scheduling—the strategy used in Figure 5.8(c). In order to perform ASAP scheduling, instead, one would visit the cluster in topological order, and pick the least significant bit that is both set, and placed on a row below all the predecessors. Exhaustive search can also be performed by adding backtracking to the ASAP strategy.

The number of elements needed on each row is computed by summing the number of computation nodes allocated to the row, and the pass-through nodes that are added between the rows. If a_j is the row on which node j is allocated, first of all s_j, the maximum row for all the successors of j, is computed. Then, a pass-through node is needed for rows in the range (a_j, s_j). This can also be expressed as a bitmask, by evaluating $(1 \ll s_j) - (1 \ll (a_j + 1))$.

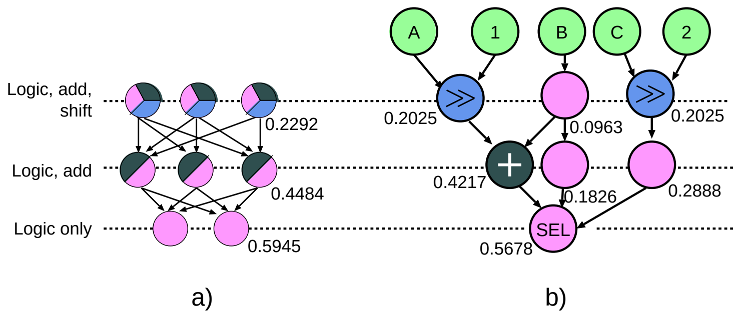
5.2.4 Candidate retiming

The EGRA architecture may allow a sequence of cells to execute in the same cycle, as long as the total critical path delay is shorter than the cycle time. This allows creation of relatively complex combinational structures and improves the number of instructions per cycle. However, because of this the compiler has the additional task of computing the run-time delays of the EGRA, in order to optimally insert registers.

Figure 5.9(a) shows latency data for the RAC's components. This can be computed beforehand, for example with Synopsys Design Compiler, but it is actually only a worst-case value for a particular RAC design. Since programming is done in advance, it is known that the switching activity of some arithmetic or logic components will not affect the outputs; if these components are on the critical path, a RAC's delay at run-time will be better than the value computed by Design Compiler.

For example, the adder is on the critical path of a full-featured ALU (supporting logic, add/subtract and shift/rotate operations). However, in the specific configuration of Figure 5.9(b), we know that the switching activity on the first row's adders will not influence the outputs, and therefore that the adders' latency will not affect the critical path! Since the compiler knows how the RACs are programmed, it should be able to obtain a refined estimate for each RAC taking part in the computation.

To achieve this objective, the compiler includes a simple model of the RAC's structure. This model splits the delays in *operation delays* (that depend on a



opcode	delay
>>	0.1062
+	0.1329
SEL	0.0764

row	mux delay
logic	0.0697
full	0.0963
other	0.0863

c)

Figure 5.9. Computing the critical path delay. a) Synopsys Design Compiler is used to compute the critical path delay of the ALU components and of the multiplexer between rows. b) Knowing the actual opcodes allows the compiler to produce a better estimate of the critical path delay. c) The model that the compiler uses considers operation and multiplexer delays separately.

node's opcode) and *multiplexer delays* (that only depend on the cell template). These delays can also be computed with Design Compiler, based on the delays for various pieces of the RAC datapath; they can then be tabulated as in Figure 5.9(c) and included in the compiler. In the example of Figure 5.9(b), the configuration-specific delay is 4.5% better than the worst-case delay of Figure 5.9(a).

After the candidate graph is partitioned into clusters, this model is applied to each cluster. After the delays are computed, the compiler does not need to know how ALUs are connected inside each cluster. Therefore, nodes in the same partition can be collapsed.

A retiming algorithm is then run on the candidate to insert registers between clusters, using a user-provided cycle time. Since data-flow graphs are acyclic, we can use a simple, linear time algorithm [Calland et al., 1998] to do so. Finally, I/O operations between cells and register file are scheduled according to the scheme of Pozzi and Ienne [2005]; this is an *NP*-complete problem (as we will prove in Section 5.3), but the approximate algorithm of Verma et al. [2007] achieves good results.

It is important to note that retiming must be run *after* partitioning, because registers cannot be inserted in the middle of a cluster (i.e., in the middle of a RAC): all cells participating to the computation must execute in less than a single cycle, and their execution must lie within a single cycle. This is easily achieved by running retiming after the clusters have been collapsed to a single node.

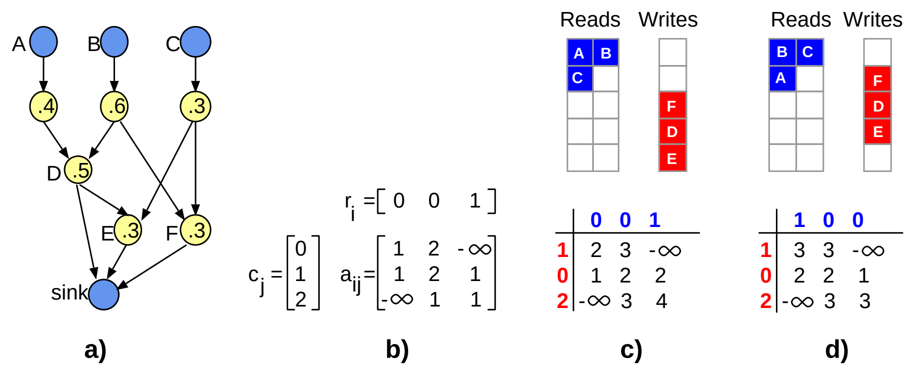

A
B
C
.4
.6
.3
D
.5
E
.3
F
.3
sink
a)
$r_i = [\,0\ \ 0\ \ 1\,]$
$c_j = \begin{bmatrix}0\\1\\2\end{bmatrix}$
$a_{ij} = \begin{bmatrix}1 & 2 & -\infty\\1 & 2 & 1\\-\infty & 1 & 1\end{bmatrix}$
b)
Reads
Writes
A B
C
F
D
E
$\begin{array}{c|ccc} & 0 & 0 & 1\\ \hline 1 & 2 & 3 & -\infty\\ 0 & 1 & 2 & 2\\ 2 & -\infty & 3 & 4\end{array}$
c)
Reads
Writes
B C
A
F
D
E
$\begin{array}{c|ccc} & 1 & 0 & 0\\ \hline 1 & 3 & 3 & -\infty\\ 0 & 2 & 2 & 1\\ 2 & -\infty & 3 & 3\end{array}$
d)

Figure 5.10. The I/O scheduling problem: a) a data-flow graph whose I/O constraints exceeds the bandwidth of the register file; b) its integral critical path delay matrix A, and the row/column vectors corresponding to a register file with 2 read ports and 1 write port; c) a permutation of R and C giving a non-optimal solution; the permutation of R gives the order of inputs, while the permutation of C gives outputs in reverse order; d) a permutation of R and C corresponding to an optimal solution.

5.3 The complexity of I/O scheduling

As we mentioned multiple times in this chapter, *I/O serialization* is an effective strategy to implement instruction set extensions with large numbers of inputs or outputs. These schemes maps inputs and outputs on the available register file ports by distributing register file accesses over more than one cycle. The new problem that arises is then to find a valid serialization for I/O between the processor and the custom functional unit, according to a given constraint on the number of register file accesses per cycle. Figure 5.10(a) shows a dataflow graph with 3 inputs and 3 outputs, while Figures 5.10(c) and 5.10(d) show two of its possible schedules for a register file with 2 read ports and 1 write port. The former takes five cycles, while the latter is optimal and takes only four.

In this section we will briefly analyze the I/O scheduling problem, and then prove its *NP*-completeness in Section 5.3.2.

5.3.1 Problem formalization

I/O scheduling was presented first by Pozzi and Ienne [2005] and solved there using brute force. The solver enumerated exhaustively all possible schedules of the inputs, looking for the one which exhibited the smallest latency. This allows to minimize not only the latency of the ISE (i.e. the makespan of the schedule), but also the number of registers used.

On the other hand, the complexity of this approach is prohibitive. If N_{in} is the number of inputs in the ISE, and N_{read} is the number of register file read ports, the number of cases to be enumerated is

$$\begin{aligned} & \binom{N_{\text{in}}}{N_{\text{read}}}\binom{N_{\text{in}}-N_{\text{read}}}{N_{\text{read}}}\cdots\binom{N_{\text{read}}}{N_{\text{read}}} \\ &= \frac{N_{\text{in}}!}{N_{\text{read}}!^{N_{\text{in}}/N_{\text{read}}}} \\ &= O\left(\frac{N_{\text{in}}!}{N_{\text{read}}{}^{N_{\text{in}}}}\right) = O\left(\left(\frac{N_{\text{in}}}{N_{\text{read}}}\right)^{N_{\text{in}}}\right) \end{aligned} \tag{5.1}$$

Evaluating each of these cases is relatively cheap (linear in the number of nodes in the ISE), but exhaustive search clearly does not scale; for $N_{\text{in}} = 14$ and $N_{\text{read}} = 2$, the possible schedules are already half a billion, and 81 billion for $N_{\text{in}} = 16$. In fact, this is the reason why Verma et al. [2007] propose a heuristic algorithm of polynomial complexity for this problem.

In the remainder of this section, we adopt their formulation of I/O scheduling as a matrix problem, which we present shortly. The delays between the inputs and outputs of the circuit are embodied by a matrix A of *integral critical path delays* between inputs and outputs, and the number of registers is defined by two vectors R and C[3]:

$$r_i = \left\lfloor \frac{i}{N_{\text{read}}} \right\rfloor \qquad c_j = \left\lfloor \frac{j + N_{\text{write}} - 1}{N_{\text{write}}} \right\rfloor \tag{5.2}$$

Figure 5.10(b) shows the matrix formulation of I/O scheduling for the dataflow graph of Figure 5.10(a). As in the picture, elements of A will be set to $-\infty$ in case there is no path between an input and an output.

The problem is then the following:

Problem 6 (*I/O scheduling*): Given a maximum number of inputs and outputs that can be scheduled in any cycle (respectively N_{read} and N_{write}), let R and C be defined as in equation (5.2). Then, given an $N_{\text{in}} \times N_{\text{out}}$ matrix A, find permutations π and σ respectively of $\{0, 1, \ldots, N_{\text{in}} - 1\}$ and $\{0, 1, \ldots, N_{\text{out}} - 1\}$, such that

[3]We assume 0-based indices in the rest of the chapter.

the following expression is minimized:

$$\lambda = \max_{i,j}(r_{\pi_i} + a_{ij} + c_{\sigma_j}) \tag{5.3}$$

The outcome λ of the minimization is the latency of the resulting ISE; inputs will be scheduled at cycle r_{π_i} and outputs at cycle $\lambda - c_{\sigma_j}$. Figures 5.10(c) and 5.10(d) show, together with the graphical representation, two numeric solutions for the input data of Figure 5.10(b), both numerically and graphically. It is easy to see that, consistently with the graphical representation, the first of the two numerical solutions has higher λ than the second.

Verma reports that their polynomial solution to this problem always found the optimal latency for the cases in which brute-force search would terminate; however, they did not have a proof of optimality. In fact, in the remainder of this section we will prove the *NP*-completeness of problem 6.

5.3.2 NP-completeness proof

First of all, we prove that I/O scheduling is in *NP*. We then introduce the decision version of the problem:

Problem 7 (*Decision version of I/O scheduling*): Given a maximum number of inputs and outputs that can be scheduled in any cycle (respectively N_{read} and N_{write}), let R and C be defined as in equation (5.2). Then, given an $N_{\text{in}} \times N_{\text{out}}$ matrix A, and a latency λ, find whether or not there exist permutations π and σ respectively of $\{0, 1, \ldots, N_{\text{in}} - 1\}$ and $\{0, 1, \ldots, N_{\text{out}} - 1\}$, such that the following expression is true:

$$\max_{i,j}(r_{\pi_i} + a_{ij} + c_{\sigma_j}) < \lambda \tag{5.4}$$

The two permutations π and σ are a certificate for problem 7. Furthermore, their size is $O\left(N_{\text{in}} + N_{\text{out}}\right)$, while the size of the problem input is $O\left(N_{\text{in}}N_{\text{out}}\right)$. Therefore, the problem admits a polynomial certificate and is in *NP*.

In order to prove the other direction, we reduce a particular flowshop scheduling problem to I/O scheduling. The scheduling problem we use is 2-machine flowshop with delays and unit job lengths (denoted shortly as *F2UD*), and has been proved to be strongly *NP*-complete by Yu [2004; 1996].

Problem 8 (*F2UD*): Given two machines M_1 and M_2, n jobs j ($j = 0, 1, \ldots, n-1$) whose execution takes 1 unit of time on M_1 and 1 unit of time on M_2, and a delay vector l_j, we define:

- t_{1j} as the time at which the first half of job j is scheduled. For any two jobs j and k, $t_{1j} \neq t_{1k}$.
- t_{2j} as the time at which the second half of job j is scheduled. For any two jobs j and k, $t_{2j} \neq t_{2k}$. Furthermore, for any job j, $t_{2j} \geq t_{1j} + l_j + 1$.
- $T_j = t_{2j} + 1$ as the completion time of job j.

The problem is then to find a schedule for the jobs that minimizes the makespan

$$T = \max_j T_j \tag{5.5}$$

We reduce F2UD to I/O scheduling with $N_{\text{read}} = N_{\text{write}} = 1$. Thus, we prove strong *NP*-completeness of I/O scheduling *even for* $N_{\text{read}} = N_{\text{write}} = 1$. In this case, equation (5.3) simplifies to the following:

$$\lambda = \max_{i,j}(\pi_i + a_{ij} + \sigma_j) \tag{5.6}$$

because $r_i = i$ and $c_j = j$. Furthermore, we set $N_{\text{in}} = N_{\text{out}} = j$, $a_{jj} = l_j + 1$, and $a_{ij} = -\infty$ everywhere except on the main diagonal. This further reduces equation (5.6) to

$$\lambda = \max_i(\pi_i + l_i + 1 + \sigma_i) \tag{5.7}$$

Given π and σ that correspond to an optimal solution (i.e., to a minimal value of λ), we can derive the scheduling times at M_1 and M_2 from π and σ respectively, by setting i.e. $t_{1j} = \pi_j$ and $t_{2j} = \lambda - \sigma_j$. This is a valid solution for 2-machine flowshop scheduling, because

$$\begin{aligned} t_{2j} &= \lambda - \sigma_j \\ &= \max_i(\pi_i + l_i + 1 + \sigma_i) - \sigma_j \\ &\geq \pi_j + l_j + 1 + \sigma_j - \sigma_j \\ &= \pi_j + l_j + 1 \\ &= t_{1j} + l_j + 1 \end{aligned} \tag{5.8}$$

This solution has makespan $T = \lambda + 1$, and is also an optimal solution. Suppose there existed a solution of problem 8 with a makespan $T' < T$. We can assume without loss of generality that the t'_{2j} vector in the solution is a permutation of $\{T'-n,\ldots,T'-2,T'-1\}$—if this was not the case, it would be possible to shift the execution of the second half of one or more jobs in order to satisfy this condition. Likewise, we can assume that the t'_{1j} vector in the solution is a permutation of $\{0,1,\ldots,n-1\}$, by anticipating the execution of some jobs on M_1 if this was not the case.

Then, by setting $\pi'_j = t'_{1j}$, and $\sigma'_j = T'-1-t'_{2j}$, we have a solution of I/O scheduling with latency:

$$\begin{aligned}
\lambda' &= \max_j(\pi'_j + l_j + 1 + \sigma'_j) \\
&= \max_j(t'_{1j} + l_j + T' - t'_{2j}) \\
&= T' + \max_j(t'_{1j} + l_j - t'_{2j}) \\
&\leq T' - 1 \\
&< T - 1 = \lambda
\end{aligned} \tag{5.9}$$

This implies $\lambda' < \lambda$, which contradicts the optimality of the I/O schedule given by π and σ.

Chapter 6

Building a compiler for customizable processors

The techniques shown in the previous chapters can be summarized like this: just like a retargetable compiler *reads a machine description* as input and uses that to generate code for different machines, a compiler for a customizable processor *generates first the machine description* depending on the input application (Figure 6.1) and then code for this specialized machine.

The specific step of generating the machine description, embodied by the selection of instruction set extensions, can be organized in the simplest possible

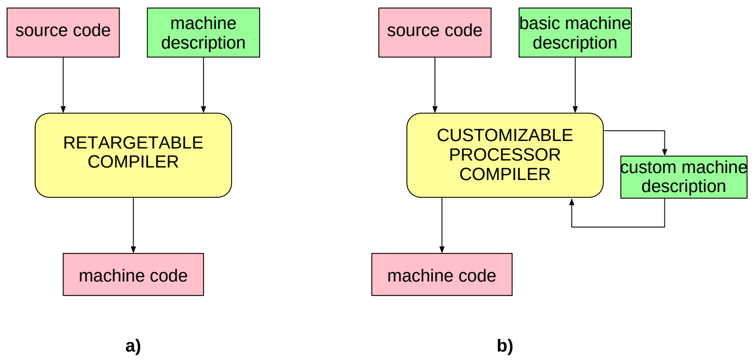


Figure 6.1. a) A retargetable compiler reads in a machine description and generates code for it. b) A compiler for a customizable processor can generate the description of the best machine for a given application, and then produce code for it.

case as described in Section 2.3. However, more complex organizations can arise from the following observation: 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 in this new scenario?

In this final chapter we argue that there is indeed a need to revisit traditional compiler transformations, noticing that techniques for automated ISE selection can expose to the compiler information about which optimizations will benefit ISE selection and which will be neutral or detrimental. We will first examine our motivation for this work in Section 6.1, and then propose a problem formalization and solution technique in Section 6.2.

6.1 Motivation and problem formulation

Figure 6.2(a) shows a C function that updates a 16-bit CRC starting from an input byte. The compiler can simplify the code a good deal using techniques

```
uint16_t crc (uint16_t crc, uint8_t 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)

```
uint16_t crc (uint16_t crc, uint8_t 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 6.2. a) C code for updating a 16-bit CRC; b) Same code after algebraic simplification, jump threading and value range propagation.

such as algebraic simplification, jump threading and value range propagation, as shown in Figure 6.2(b). Still, this is not yet a good starting point for ISE search since identification techniques, as shown in the previous chapters, usually identify extensions only within a single basic block level. Therefore, only small sections in this code could be identified as custom instructions.

However, this snippet hides very high potential. By performing if-conversion *and* total loop unrolling prior to ISE identification, the CRC function is transformed into a single basic block. In other words, this makes the computation purely combinational, so that all of it can be included in a single custom instruction.

Of course, 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 extensible processor, the code will actually be compiled down to a single instruction and all intermediate results are transformed into wires, so that no register pressure and no cache pollution problems arise.

6.2 Optimization in a compiler for customizable processors

In the previous chapters the chosen intermediate representation was the data-flow graph (Section 3.1), since the search for instruction-set extensions was done primarily within a single basic block. A more generic intermediate representation howeer will be more similar to a collection of control/data-flow graphs (CDFGs), one per function, capturing 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.

We propose a technique for ISE-targeted CDFG transformations that, despite its simplicity, can catch significant optimization opportunities. By this technique, before ISE identification we transform the CDFG into a semantically equivalent one, yielding higher gain from the subsequent ISE identification phase (Figure 6.3. The problem is then reduced to finding the best set of transformations to apply, which is formalized as follows:

Problem 9 (*ISE-targeted code transformation*): Given a control/data-flow 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.,

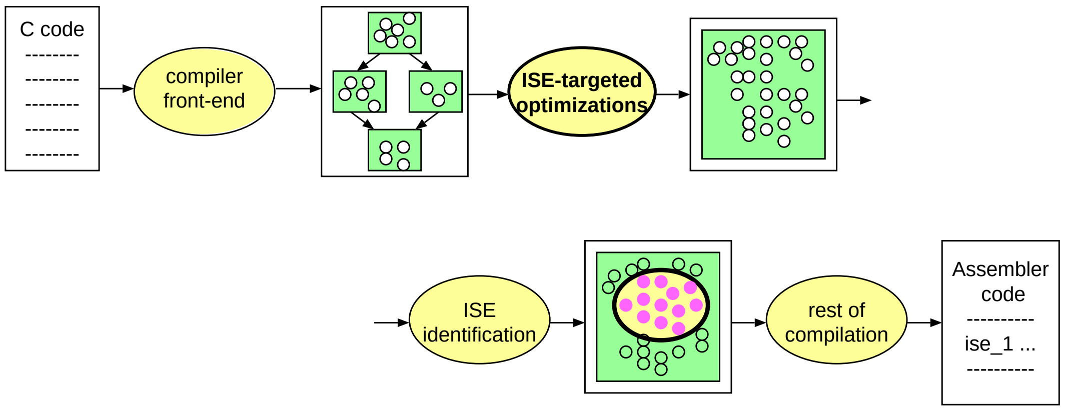


Figure 6.3. By applying ISE-targeted transformations to the code, better high-performance extensions can be found by ISE identification. Therefore, a set of possible CDFG transformations is considered, and the one yielding the best ISE is selected.

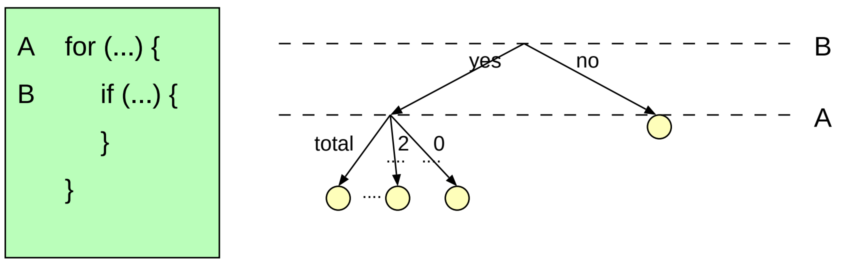
select the transformed $CDFG'$ that, when fed to the chosen ISE identification algorithm, exposes and returns the ISE with highest gain.

The peculiar point of our solution is that not only code transformations are applied in an attempt to improve the quality of the custom instructions, but the resulting extensions are examined to discard transformations that turned out to be useless. This bidirectional flow of information between the optimizations and the ISE search algorithms avoids excessive code size increase, and still allows to use traditional compiler heuristics effectively after custom instructions have been selected.

We apply this technique to two control-flow transformations, if-conversion and loop unrolling. This small set is enough to show how ISE identification can benefit greatly from specialized heuristics in a compiler targeting customizable processors.

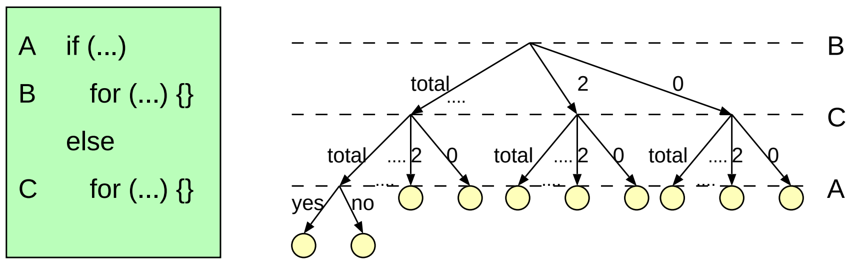
6.2.1 The transformation space

First of all, we study the meaning of the transformation space, depicted in Figure 6.4. Once a set of possible transformation types has been defined, different spots in the intermediate representation can be identified where one of the



```
A   for (...) {
B       if (...) {
        }
    }
```

a)



```
A   if (...)
B      for (...) {}
    else
C      for (...) {}
```

b)

Figure 6.4. 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, but different leaves may expose a different set of ISE.

transformations can potentially be triggered. In the example of Figure 6.4(a), modeled on Figure 6.2, there are 2 such spots, labeled A and B.

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. Leaves of the tree represent CDFGs that are semantically equivalent to the initial one and can be obtained by applying the different transformations. The space thus appears to grow very fast 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 set of transformation types we have chosen.

Of course an obvious solution to problem 9 is to exhaustively explore the transformation space, applying ISE identification 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 identification to points corresponding to huge basic blocks, perhaps resulting from total unrolling—remember that the complexity of enumeration and covering, grows relatively fast with the number of nodes in the basic block. 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*. Based on the outcome, it is able to select the one point that is the solution to problem 9.

6.2.2 Finding the best transformation set

In order to do so, we propose the following steps:

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 ISE identification to the CDFG corresponding to the chosen point.

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

This can be understood better in Figure 6.5. 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 unrolled CDFG. The rules leading to this decision will now be explained.

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

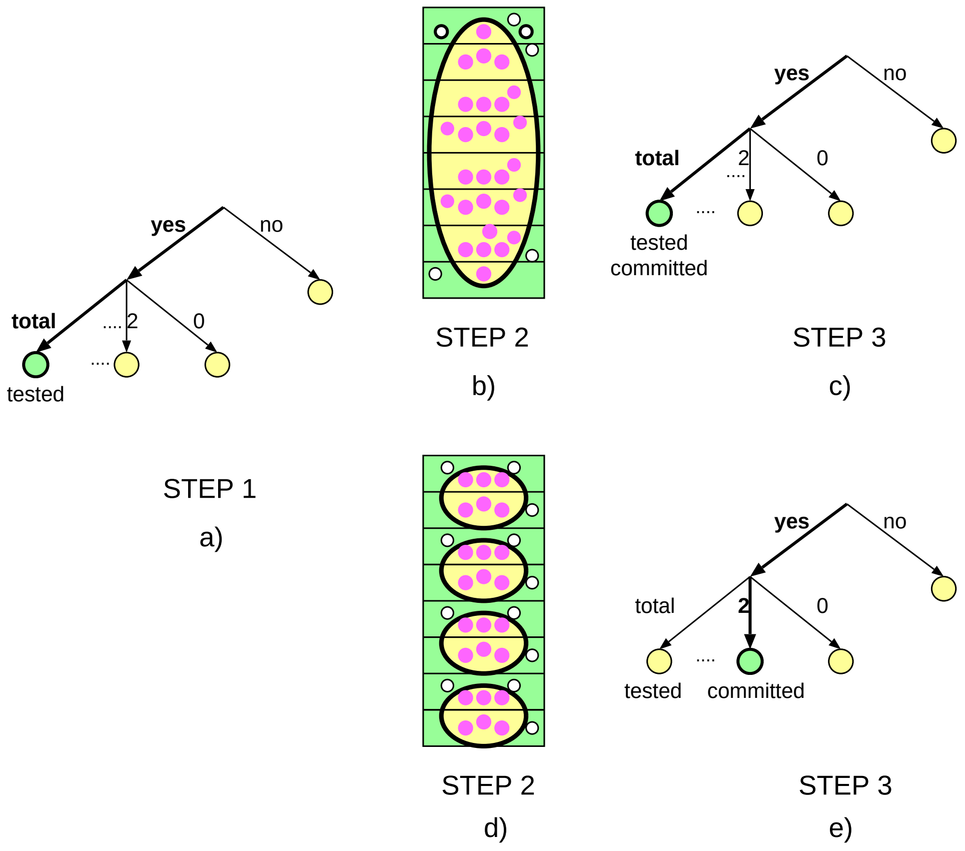


Figure 6.5. A pictorial description of the algorithm's operation. a) In step 1, a leaf is chosen as the single CDFG that is searched for instruction set extensions. b) In step 2, one identified ISE is analyzed and found to contain nodes from the first and last iteration c) step 3 therefore detects that loop unrolling is exploited, and commits the total-unrolling choice. Since edges to the multiplexer (not shown) are also included in the ISE, if-conversion is committed too.
d) and e) In this case the ISE only spans 2 iterations instead, therefore only two-iteration unrolling is committed. The winner in this case is not the point on which ISE selection was attempted.

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, as in Figure 6.6. 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.

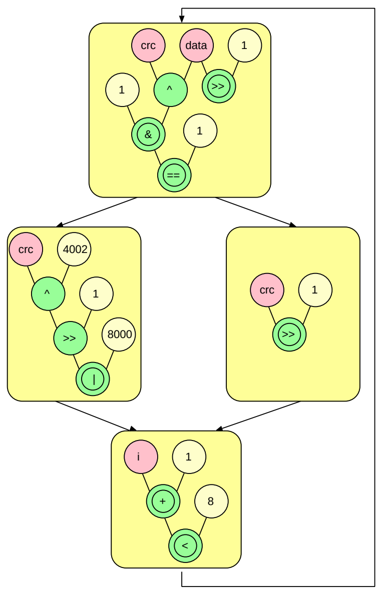
Unrolling. Unrolling can expose ISE in two ways. A single, disconnected multiple-output ISE can span across multiple iterations, performing them in parallel (we call this an opportunity for *horizontal* unrolling) or, if loop carried dependences exist, a single connected ISE can chain multiple iterations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

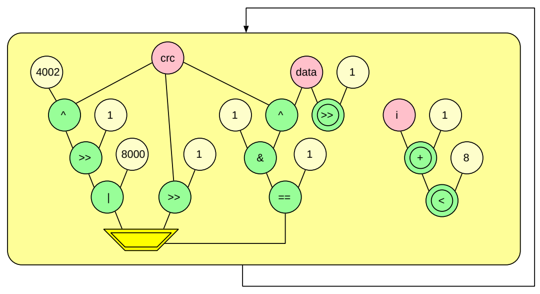
For spots that consist of a single basic block, with a self-loop and only one additional outgoing edge, the whole loop body may be covered with a single ISE. The compiler checks if this is possible, and computes the highest value of the unrolling factor n for which this condition holds. The following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

[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.



a)



c)

```
⟨bb1⟩
  $i_1 = \phi(0, i_2)$
  $data_2 = \phi(data_1, data_3)$
  $crc_2 = \phi(crc_1, crc_5)$
  $x_1 = (data_2 \mathbin{\hat{}} crc_2) \mathbin{\&} 1$
  $data_3 = data_2 >> 1$
  if $(x_1 == 1)$ goto ⟨bb2⟩ else goto ⟨bb3⟩
⟨bb2⟩
  $crc_3 = ((crc_2 \mathbin{\hat{}} 0x4002) >> 1) \mid 0x8000$
  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 $crc_5$
```

b)

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

d)

Figure 6.6. a-b) CDFG and SSA representation of the crc example after the compiler's scalar optimization passes; c-d) CDFG and SSA after if-conversion replaced the region by a single basic block.

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 6.6(c) 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 6.6(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 $exit_1$—all of 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:

$$\begin{aligned} tot_{in} &= n|H_{in}| + |V_{in}| \\ tot_{out} &= n|H_{out}| + |V_{out}| \end{aligned}$$

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

$$\begin{aligned} tot_{in} &= n|H_{in}| + |V_{in}| - |T_{in}| \\ tot_{out} &= n|H_{out}| + |V_{out}| - |T_{out}| \end{aligned}$$

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 6.6(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.

Applying transformations on the CDFG. Both if-conversion and unrolling are easy to perform on SSA-form CDFGs.

If-conversion merges the header with the conditional blocks. New nodes are created for each of the junction block's ϕ functions; their definition at the bottom of the fused region consists of a `?:` expression representing a multiplexer, and replaces the two arguments of the ϕ function that correspond to the conditional blocks. Possibly, the junction will enter the fused region as well (if the conditional blocks are its only predecessors). If this is the case, ϕ functions will be eliminated completely.

To perform unrolling, instead, multiple copies of the loop are created and juxtaposed in the same graph. Then, edges are created 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.

A single ISE may span multiple iterations, so we have to ensure that loop unrolled n times will run for a number of iterations that is a multiple of n. In our implementation, this requires the loop test to be a comparison between a basic induction variable and a loop invariant.

In either case, fusion will enable outputs of some DFGs to be merged with inputs of other graphs, creating new edges in the fused graph; furthermore, variables that are not live at the exit of the region may be removed from the list of outputs.

The compiler tracks new data-flow edges created by the transformations. After ISE search, if an instruction includes a newly created edge, the compiler finalizes the transformation by committing it on its SSA-based intermediate representation.

6.3 Experimental results

The ISE-targeted unrolling and if-conversion heuristics we presented were implemented in the same toolchain used for Section 4.4, using GCC and SimpleScalar/ARM.

Our extension of SimpleScalar can accept the definitions of up to seven ISEs, each with up to four inputs and up to two outputs; ISE descriptions are written in C and dynamically linked to the simulator. The compiler's machine description was modified to generate assembly code for the extensions; the actual meaning of these instructions, however, is obviously defined only during the compilation of user programs.

After ISE selection, the compiler finds the selected occurrences in its intermediate representation, and modifies it to use extended instructions instead of software evaluation. The compiler also emits C code for the instructions, which SimpleScalar will dynamically link to.

	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 6.1. 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
blowfish	no	no	yes
des	no	yes	no
sha	no	yes	yes
bitcount	no	yes	yes
crc	yes	yes	no

Table 6.2. Transformations that trigger during compilation of the benchmarks.

The benchmarks were compiled with GCC's `-O2` optimization level, enabling feedback-directed optimization and inter-procedural analysis. Note that GCC's usual unrolling heuristics are not disabled, because ISEs cause code size reduction and may cause them to trigger more often.

We tuned SimpleScalar to match the architecture of the XScale, a single-issue processor with in-order execution (6.1). The latency and area of custom instructions were estimated using data from Artisan UMC $0.18\mu m$ technology and simple bitwidth analysis techniques. While memory operations are not included in extended instructions, read-only global arrays are an exception as they

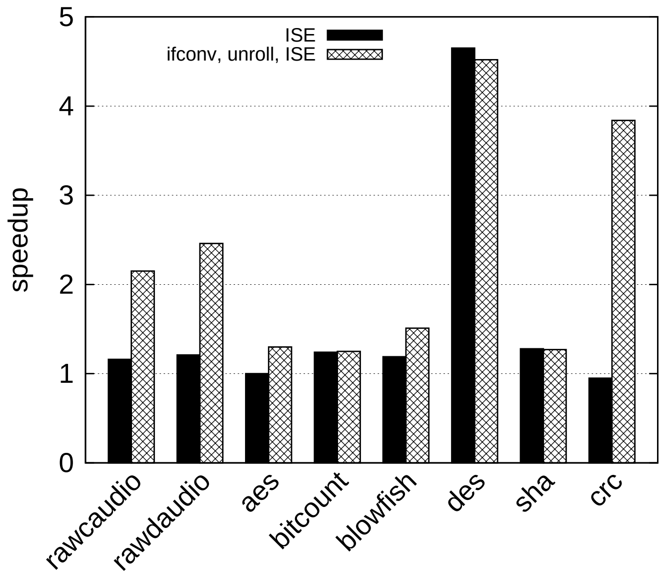


Figure 6.7. Speedup obtained on customizable processors, without (solid), and with CFG transformations (hatched).

can be converted to hardware lookup tables in the ISE. Area estimates from the compiler take into account bits that are always 0 or 1 in the tables.

Figure 6.7 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 6.2 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 6.1 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

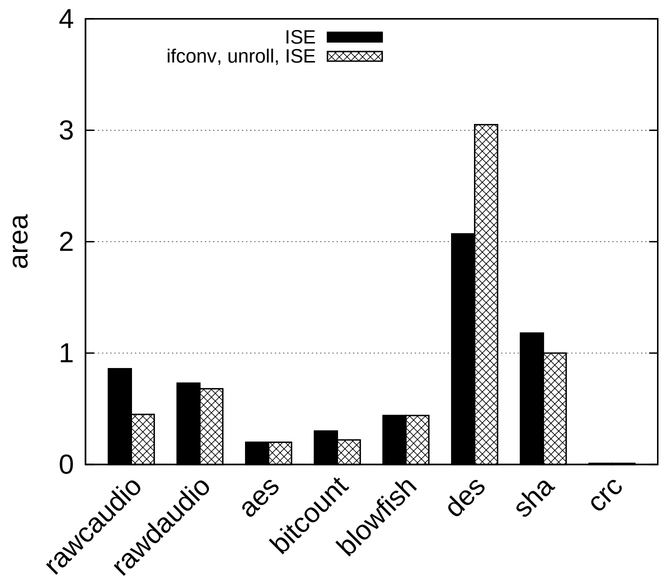


Figure 6.8. Area cost of the AFUs that were generated without (solid), and with CFG transformations (hatched).

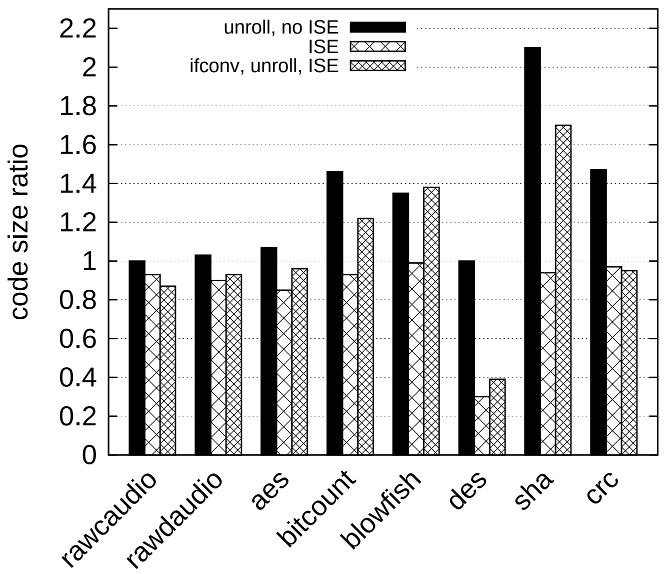


Figure 6.9. Code size on a non-customizable processor with unrolling, and on a customizable processor without and with CFG transformation. 1.0 = non-customizable processor, no unrolling.

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 6.8 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.

While not directly related to ISE-targeted optimization techniques we describe, one other interesting observation is possible. ISEs cause code size reduction, so the compiler may choose to unroll more loops after ISEs are selected than it would have before; sometimes, the same loops that were already unrolled by our pass are expanded further. Indeed, the plots in figure 6.9 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.

Chapter 7

Conclusions

This work's hypothesis was that a generalization of existing research results (the *framework*), together with the specification of the common parts of these solutions (the *interfaces*) is a powerful way to solve a problem in an easily adapted way. Adaptation means being able to use the solution in a different context—either targeting a new technology, or improving some of the underlying algorithms—while keeping most of the implementation in place.

The claim, then, is that this hypothesis can be applied in the context of compilation for extensible embedded processors. The various parts of the proposed framework, which is defined based on a survey of existing work, were analyzed in depth and adapted to different technologies and uses; well-known algorithms were analyzed or generalized, and new ones were developed whenever needed.

Regarding the first step of the framework, enumeration, we adapted a technique used for the *exact cover* enumeration problem, and produced a new algorithm for *independent set* enumeration. This algorithm is fast, easy to implement, and requires a small amount of memory to run. We also analyzed a very well-known algorithm from the state-of-the-art in depth, proving an optimal lower bound on its time complexity. The proof hints that the algorithm is fast because it considers all of the constraints and is able to prune its search based on each of them; we then extended the idea so that additional speed can be gained if the output is even more constrained.

Regarding the second step of the framework, covering, we noticed how interprocedural covering can be easily reduced to the intraprocedural version. Based on this, we analyzed the relative merits of greedy and optimal solutions of the intraprocedural problem. We also introduced isomorphism-aware search, showing that information on recurrence on the same ISE can effectively improve the quality of the extensions generated for more difficult benchmarks.

Finally, we noticed how the proposed framework is not restricted to searching multiple extensions in a program; rather, it can also be used to find a mapping for a single extension to a coarse-grained reconfigurable architecture (and in particular the EGRA, whose design is outlined in Chapter 5).

Is there any other interesting work to do on compilation for extensible processors? Sure. The last chapter of the thesis showed that searching for the best set of ISEs provides only half of the solution; it is akin to having the best instruction selection pass in a traditional compiler, but no loop optimizations, no conditional execution, no alias analysis. Compilers like this exist [Thompson, 1990], but extracting performance from a modern processor requires all those optimizations and more.

Existing compilation techniques, especially loop-oriented ones such as software pipelining, hint at which transformations need to be revisited so that they can help finding even better instruction-set extensions. This is especially important for accelerators that can execute entire loops autonomously, such as most coarse-grained reconfigurable accelerators. As researchers from various institutions start to appreciate the importance of having a single tool capable to target different CGRAs, it is our hope that the ideas presented in this thesis help with the design of such a framework.

Appendix A

Terminology

Non-direct graphs.

V	Set of vertices of the graph
E	Set of edges of the graph
u, v, w	Vertices of the graph
(u, v)	An edge between u and v
$u \to v, v \to u$	An edge between u and v; used when a path is being followed
$N(u)$	Neighbors of a vertex u

Direct graphs.

V	Set of vertices of the graph
E	Set of edges of the graph
u, v, w	Vertices of the graph
$u \to v$	An edge from u to v
pred(u)	Immediate predecessors of u, i.e. $\{v \in V : v \to u \in E\}$
succ(u)	Immediate successors of u, i.e. $\{v \in V : u \to v \in E\}$
Pred(u)	All predecessors of u, i.e. nodes from which there is a path to u
Succ(u)	All successors of u, i.e. nodes reachable from u

Pseudo-code.

x.field	Accessing *structs*
$\triangleright$ text. . .	Comments

Appendix B

Publication reference

Part of the material in this thesis was already published in conference and journal publications, as well as in technical reports. This appendix details the correspondence between this thesis and previous publications.

Section 2.3: 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*, pages 334–41, Montréal, Canada, July 2007.

Section 3.1: 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*, pages 1331–36, Nice, France, Apr. 2007.

Section 3.3: P. Bonzini and L. Pozzi. On the complexity of enumeration and scheduling for extensible embedded processors. Technical Report 2008/07, University of Lugano, Lugano, Switzerland, Dec. 2008. **URL** `http://www.inf.unisi.ch/file/pub/46/bonzini-pozzi-2008-07.pdf`.

Sections 4.3 and 4.4: P. Bonzini and L. Pozzi. A retargetable framework for automated discovery of custom instructions (*cit.*).

P. Bonzini and L. Pozzi. Recurrence-aware instruction set selection for extensible embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(10), Oct. 2008.

Chapter 5 except Section 5.3: G. Ansaloni, P. Bonzini, and L. Pozzi. Design and architectural exploration of expression-grained reconfigurable arrays. In

Proceedings of the 6th Symposium on Application Specific Processors, pages 26–33, Anaheim, CA, June 2008.

P. Bonzini, G. Ansaloni, and L. Pozzi. Compiling custom instructions onto expression-grained reconfigurable architectures. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 51–60, Atlanta, GA, Oct. 2008.

Section 5.3: P. Bonzini and L. Pozzi. On the complexity of enumeration and scheduling for extensible embedded processors (*cit.*).

Chapter 6: 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*, pages 242–52, Seoul, South Korea, Oct. 2006.

Bibliography

M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi [2006]. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 363–68. European Design and Automation Association 3001 Leuven, Belgium, Mar. 2006.

A. V. Aho, R. Sethi, and J. D. Ullman [1988]. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1988.

C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami [1999]. A DAG based design approach for reconfigurable VLIW processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 778–79, Mar. 1999.

Altera Corp. [2002]. Custom instructions for the Nios embedded processor. Application Note AN-188-1.1. San Jose, CA, Apr. 2002.

G. Ansaloni, P. Bonzini, and L. Pozzi [2008]. Design and architectural exploration of expression-grained reconfigurable arrays. In *Proceedings of the 6th Symposium on Application Specific Processors*, pages 26–33, Anaheim, CA, June 2008.

G. Ansaloni, P. Bonzini, and L. Pozzi [2009]. Heterogeneous coarse-grained processing elements: a template architecture for embedded processing acceleration. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, Mar. 2009. To appear.

K. Atasu, L. Pozzi, and P. Ienne [2003]. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pages 256–61, Anaheim, CA, June 2003.

P. Biswas, N. Dutt, P. Ienne, and L. Pozzi [2006]. Automatic identification of application-specific functional units with architecturally visible storage. In

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pages 212–17, Munich, Germany, Mar. 2006.

P. Bonzini and L. Pozzi [2006a]. Polynomial-time subgraph enumeration for automated instruction set extension. Technical Report 2006/07, University of Lugano, Lugano, Switzerland, Dec. 2006. **URL** `http://www.inf.unisi.ch/file/pub/15/bonzini-pozzi-2006-07.pdf`.

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

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

P. Bonzini and L. Pozzi [2007b]. A retargetable framework for automated discovery of custom instructions. In *Proceedings of the 18th International Conference on Application-specific Systems, Architectures and Processors*, pages 334–41, Montréal, Canada, July 2007.

P. Bonzini and L. Pozzi [2008a]. On the complexity of enumeration and scheduling for extensible embedded processors. Technical Report 2008/07, University of Lugano, Lugano, Switzerland, Dec. 2008. **URL** `http://www.inf.unisi.ch/file/pub/46/bonzini-pozzi-2008-07.pdf`.

P. Bonzini and L. Pozzi [2008b]. Recurrence-aware instruction set selection for extensible embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(10), Oct. 2008.

P. Bonzini, G. Ansaloni, and L. Pozzi [2008]. Compiling custom instructions onto expression-grained reconfigurable architectures. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 51–60, Atlanta, GA, Oct. 2008.

P. Brisk, A. Kaplan, and M. Sarrafzadeh [2004]. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the 41st Design Automation Conference*, pages 395–400, San Diego, USA, June 2004.

P. Brisk, J. Macbeth, A. Nahapetian, and M. Sarrafzadeh [2005]. A dictionary construction technique for code compression systems with echo instructions. In *Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 105–14, New York, NY, USA, June 2005. ACM Press.

P. Y. Calland, A. Mignotte, O. Peyran, Y. Robert, and F. Vivien [1998]. Retiming DAG's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1319–25, Dec. 1998.

A. Chattopadhyay, X. Chen, H. Ishebabi, R. Lupers, G. Ascheid, and H. Meyr [2008]. High-level modelling and exploration of coarse-grained re-configurable architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1334–39, Munich, Germany, Mar. 2008.

X. Chen, D. L. Maskell, and Y. Sun [2007]. Fast identification of custom instructions for extensible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):359–68, Feb. 2007.

N. Clark, H. Zhong, and S. Mahlke [2003]. Processor acceleration through automated instruction set customization. In *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 129–40, San Diego, CA, Dec. 2003.

N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner [2004]. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO 37: Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Washington, DC, USA, Dec. 2004. IEEE Computer Society.

N. Clark, A. Hormati, S. Mahlke, and S. Yehia [2006]. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 147–57, Seoul, South Korea, Oct. 2006.

J. Cong and Y. Ding [1994]. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, Jan. 1994.

J. Cong, Y. Fan, G. Han, and Z. Zhang [2004]. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004*

ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, pages 183–89, Monterey, CA, Feb. 2004.

L. Cordella, P. Foggia, C. Sansone, and M. Vento [2004]. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–72, Oct. 2004.

M. L. Corliss, E. C. Lewis, and A. Roth [2003]. DISE: A programmable macro engine for customizing applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, volume 0, pages 362–73, Los Alamitos, CA, USA, June 2003. IEEE Computer Society.

O. Coudert [1997]. Solving graph optimization problems with ZBDDs. In *Proceedings of the European Conference on Design and Test*, pages 224–28, Mar. 1997.

D. Eppstein [2005]. All maximal independent sets and dynamic dominance for sparse graphs. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 451–59, Philadelphia, PA, USA, Jan. 2005. Society for Industrial and Applied Mathematics.

M. Galanis, G. Theodoridis, S. Tragoudas, and C. Goutis [2006]. A high-performance data path for synthesizing DSP kernels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1154–62, June 2006.

M. R. Garey and D. S. Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski [2006]. GrGen: A fast SPO-based graph rewriting tool. In *Proceedings of the 3rd Internatial Conference on Graph Transformations*, Natal, Brazil, Sept. 2006.

S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer [1999]. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.

S. W. Golomb and L. D. Baumert [1965]. Backtrack Programming. *Journal of the ACM (JACM)*, 12(4):516–24, 1965.

D. Goodwin and D. Petkov [2003]. Automatic generation of application specific processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 137–47, San Jose, CA, Oct. 2003.

Y. Guo, G. J. Smit, H. Broersma, and P. M. Heysters [2003]. A graph covering algorithm for a coarse-grain reconfigurable system. In *Proceedings of the 2003 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 199–208, New York, NY, USA, July 2003. ACM.

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

R. Hartenstein [2001]. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 642–49, Mar. 2001.

S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao [1997]. The Chimaera reconfigurable functional unit. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, Napa Valley, CA, Apr. 1997.

J. R. Hauser and J. Wawrzynek [1997]. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Napa Valley, CA, Apr. 1997.

R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh [2002]. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):605–27, Oct. 2002.

D. E. Knuth [2000]. Dancing Links. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millenial Perspectives in Computer Science*, pages 187–214. Palgrave, Houndmills, England, 2000.

D. E. Knuth [2008]. *Fundamental Algorithms*, volume 4, pre-fascicle 1a of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2008.

J.-E. Lee, K. Choi, and N. Dutt [2002]. Mapping loops on coarse-grain reconfigurable architectures using memory operation sharing. Technical Report #02-34, Center for Embedded Computer Systems, University of California at Irvine, Sept. 2002.

S. Liao, S. Devadas, K. Keutzer, and S. Tjiang [1995]. Instruction selection using binate covering for code size optimization. In *Proceedings of the International Conference on Computer Aided Design*, pages 393–99, San Jose, CA, Nov. 1995.

S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood [2001]. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–71, Nov. 2001.

B. D. McKay [1981]. Practical graph isomorphism. *Congressus Numerantium*, 30: 45–87, 1981. **URL** `http://cs.anu.edu.au/people/bdm/nauty/`.

B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins [2002]. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 166–73, Dec. 2002.

S. Minato [1993]. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference*, pages 272–77, New York, NY, USA, 1993. ACM Press.

A. Mishchenko [2001]. An introduction to Zero-Suppressed Binary Decision Diagrams, June 2001. **URL** `http://www.eecs.berkeley.edu/~alanmi/publications/2001/tech01_zdd.pdf`.

J. Moon and L. Moser [1965]. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

H. G. Okuno, S. Minato, and H. Isozaki [1998]. On the properties of combination set operations. *Information Processing Letters*, 66(4):195–99, May 1998.

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

N. Pothineni, A. Kumar, and K. Paul [2007]. Application specific datapath extension with distributed i/o functional units. In *Proceedings of the 20th International Conference on VLSI Design*, pages 551–58, Bangalore, India, Jan. 2007.

L. Pozzi and P. Ienne [2005]. 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*, pages 2–10, San Francisco, CA, Sept. 2005.

L. Pozzi, K. Atasu, and P. Ienne [2006]. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-25(7):1209–29, July 2006.

T. G. Rauscher and A. K. Agrawala [1978]. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers*, 27(11):1006–14, Nov. 1978.

R. Razdan and M. D. Smith [1994]. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80, San Jose, CA, Nov. 1994.

C. R. Rupp [2003]. Multi-scale Programmable Array. U.S. Patent 6633181, Oct. 2003.

H. Singh, L. Ming-Hau, L. Guangming, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho [2000]. Morphosys: An integrated reconfigurable system for data-parallel computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–81, May 2000.

N. J. A. Sloane [2009]. The On-Line Encyclopedia of Integer Sequences, 2009. **URL** `http://www.research.att.com/~njas/sequences/`.

F. Somenzi [1998]. CUDD: Colorado University Decision Diagram Package, 1998. **URL** `http://vlsi.colorado.edu/~fabio/CUDD/`.

M. Stephenson, J. Babb, and S. Amarasinghe [2000]. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 108–20, Vancouver, Canada, June 2000.

F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha [2004]. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):216–28, Feb. 2004.

K. Thompson [1990]. Plan 9 C compilers. In *UNIX—The Legend Evolves. Proceedings of the Summer 1990 UKUUG Conference*, pages 41–51, 90 1990.

J. Turley [1999]. Tensilica CPU bends to designers' will. *Microprocessor Report*, 8 Mar. 1999.

A. Vahidi [2003]. JDD, a pure Java BDD and Z-BDD library, 2003. **URL** `http://javaddlib.sourceforge.net/jdd`.

A. K. Verma, P. Brisk, and P. Ienne [2007]. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 125–34, Salzburg, Austria, Oct. 2007.

E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal [1997]. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sept. 1997.

M. J. Wirthlin and B. L. Hutchings [1995]. A dynamic instruction set computer. In *Proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 99–107, Napa Valley, CA, Apr. 1995.

Xilinx Inc. [2006]. *The Virtex-5 User Guide*, Oct. 2006. **URL** `http://direct.xilinx.com/bvdocs/userguides/ug190.pdf`.

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

J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek [2008]. SPKM: A novel graph-drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 776–82, Seoul, South Korea, Jan. 2008.

W. Yu [1996]. *The Two-machine Flow Shop Problem with Delays and the One-machine Total Tardiness Problem*. PhD thesis, Eindhoven University of Technology, 1996.

W. Yu, H. Hoogeveen, and J. Lenstra [2004]. Minimizing Makespan in a Two-Machine Flow Shop with Delays and Unit-Time Operations is NP-Hard. *Journal of Scheduling*, 7(5):333–48, Oct. 2004.