Partial Evaluation for Code Generation from Domain-Specific Languages

Jia Zeng

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2007

ABSTRACT

Partial Evaluation for Code Generation from Domain-Specific Languages

Jia Zeng

Partial evaluation has been applied to compiler optimization and generation for decades. Most of the successful partial evaluators have been designed for general-purpose languages. Our observation is that domain-specific languages are also suitable targets for partial evaluation. The unusual computational models in many DSLs bring challenges as well as optimization opportunities to the compiler.

To enable aggressive optimization, partial evaluation has to be specialized to fit the specific paradigm of a DSL. In this dissertation, we present three such specialized partial evaluation techniques designed for specific languages that address a variety of compilation concerns. The first algorithm provides a low-cost solution for simulating concurrency on a single-threaded processor. The second enables a compiler to compile modest-sized synchronous programs in pieces that involve communication cycles. The third statically elaborates recursive function calls that enable programmers to dynamically create a system's concurrent components in a convenient and algorithmic way. Our goal is to demonstrate the potential of partial evaluation to solve challenging issues in code generation for domain-specific languages.

Naturally, we do not cover all DSL compilation issues. We hope our work will enlighten and encourage future research on the application of partial evaluation to this area.

Contents

List of Figures

List of Tables

To my parents and my husband

Chapter 1

Introduction

1.1   Motivation and Purpose

Partial evaluation (PE) optimizes programs by specialization. The idea is simple: consider the logic function $(x \oplus y) \vee z$. If y is always 1, we can simplify the function to $\overline{x} \vee z$. In other words, we customized a "$y = 1$" version of this function. To a partial evaluator resident in a compiler, the inputs used for specialization must be static. They can be some variables whose values are known, or even the structure of the program being compiled.

   The main purpose of this dissertation is to demonstrate the potential of partial evaluation to solve challenging issues in compiling domain-specific languages (DSLs), which are designed to be used in specific fields of programming, such as Yacc for creating parsers and Verilog for designing hardware. To illustrate how to design an effective partial evaluation technique for a specific DSL, and we demonstrate three PE techniques that address various problems during code generation from DSLs.

   By comparing DSLs to general-purpose languages (GPLs), we demonstrate some characteristics of DSLs that enable PE to work aggressively on DSLs. We compare different DSLs in the same domain, and we illustrate their special computational models and com-

```
int i = 1;                                    int i = 1;

if (i > 0) {                                  System.out.println("i = 1");
  System.out.println("i = " + i);
} else {
  System.err.println("Negative number!");
}
```

Figure 1.1: An example in Java. (a) Original code. (b) After partial evaluation.

pilation challenges, which explain why it is necessary to design specific PE techniques for a DSL instead of using existing general partial evaluators.

Partial evaluation is well known for its application to compiler optimization and compiler generation. It is also referred to as *program specialization*. A specialized program usually runs faster than the original version since the partial evaluator may restructure the program's logic and carry out part of the computation at compile time. Figure 1.1 shows a print program in Java that can be simplified by partial evaluation. Nevertheless, the specialization process is nontrivial; it may change the semantics, cause an explosion in code size, or even may not terminate. For example, if in the Java example the variable *i*'s value relies on the input, the partial evaluator may explore all possible values of *i* and hence never terminate. Therefore, most PE techniques are conservative.

General PE techniques usually involve constant propagation, loop unrolling and inlining. Yet, PE is not just a collection of these techniques; it is combination of compilation techniques, language and semantics. Compared to constant propagation, which only deals with static values of variables, PE focuses on static "properties" of a program [26]. Therefore a PE system commonly performs in-depth flow analysis and requires comprehensive knowledge of semantics. For example, knowledge of data types and bounds on variable values, which are not helpful to constant propagation, may be used for program specialization, as Consel and Khoo show [26].

Our observation is that a domain-specific language, which has a simple but concise syntax, is a suitable target for applying partial evaluation. The simplified syntax eases the analysis workload of PE and enables PE to deeply understand the computational model

underlying the program. Unfortunately, our survey of PE found no successful partial evaluator for DSLs, in part because most earlier work has focused on GPLs. Although some early work did apply PE to optimization or automatic compiler generation for DSLs, such techniques were too general to achieve any significant improvement. In contrast, PE in a restricted setting works more effectively for a specific paradigm. Each PE technique we demonstrate in this dissertation, for example, is carefully designed for the specific model of computation in the corresponding DSL. Each language's special features bring challenges as well as optimization opportunities to the compiler. Our experimental results show that PE is effective at addressing these challenges.

1.2 A Brief History of Partial Evaluation

In 1952, Kleene [54] was the first to formulate partial evaluation (PE). His $s-m-n$ theorem claims that, for an arbitrary function f with $m+n$ arguments, when the values of the first m arguments are given, there always exists a specialized function g that takes the n arguments and behaves the same. More important, he proved there is a program to construct the specialized function. His theorem was later extended to improve a program's efficiency by specialization. Lombardi and Raphael, according to Jones [47], were the first to use the term "partial evaluation" in 1964 [45].

Most of the early work applied PE to compiling and compiler generation [43, 23, 6]. At that time, the most significant benefit that compilers gained from PE was not efficiency but automation. Based on the fact that a partial evaluator is a program by itself, Futamura [39] foresaw the self-application of PE, i.e., compiler generation. He did some experiments but never proved this idea. Proof would have to wait until 1985, when Jones et al. [46] created MIX, the first practical self-applicable partial evaluator for a language of first-order recursive equations. Later, more research was conducted on improving efficiency of compiled code generated by a partial evaluator, mainly by reducing the interpretive overhead added by the evaluator. Jørgensen [48], for example, managed to compile a lazy functional lan-

guage to generate code that runs faster than the commercial tool generated code does. The boom in research on PE in 1990's prompted the first PE conference in the US: the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM).

Because of its conceptual simplicity and great automation, partial evaluation has been applied to many areas other than compilation, such as pattern matching, circuit simulation, numerical computations and computer graphics.

Traditional partial evaluators can be divided into two classes: online and offline. An online partial evaluator, which typically resides in the interpreter, relies on the concrete values computed at run time and makes decisions on the fly. Most of the early partial evaluators were online, which made them accurate but often very slow. To make specialization decisions at run time, an online partial evaluator usually embeds an interpreter in the generated program. Not being carefully optimized, this interpreter can cause an intolerable speed penalty.

An offline evaluator, in contrast, generates more general code since the program specialization relies on preprocessing results, not input values. It is usually separated into at least two stages. One is preprocessing, or binding-time analysis; the other is the specialization phase. The binding-time analyzer is responsible for collecting information, then determining whether the evaluation of an expression can be done at compile time or has to be deferred to run time. Later this information will be used to guide the specialization of the program [24]. All the PE techniques we introduce in later chapters are offline. Some partial evaluators aggressively combine these two methods to achieve best result [15], i.e., if an expression can be determined to be static at binding time, offline PE is applied, otherwise it is deferred to be treated by online PE at run time.

There are now many partial evaluators for general-purpose languages, such as C-Mix for C [40], JSpec for Java [65] and SML-mix for ML [13]. One of the most successful is Tempo, an offline C specializer developed in the Compose project at INRIA [25]. From a C program and an annotation of static inputs during specialization, Tempo performs a

sequence of analyses (alias analysis, side-effect analysis, binding time analysis, etc.) and preprocessing steps (goto elimination, function pointer elimination, etc.), then passes abstract code to specialization phases, which generate efficient code. Tempo has been applied to various domains, including operating systems, networking, graphics, etc., and proved effective. Also, some compilers for other languages [57], such as Java and C++, use Tempo as an optimizing back-end.

Although Tempo accepts most of ANSI C, there are still some complex features it cannot deal with, such as bit fields and mutually recursive structures. The alias analysis has some constraints also. These complex features of C limit the specializer's precision to some degree.

The interest in applying partial evaluation to domain-specific languages has grown in recent years. Burchett et al. [18] developed a partial evaluator that reduces the size of the dynamically changed graph size when programming in an interactive dataflow language. Edwards [33] demonstrated a program specialization that dramatically speeds up fixed-point simulation of signal processing kernels written in SystemC – another high-level hardware design language like those we compare in Chapter 2. These works reinforce our observation that partial evaluation can be very effective in optimizing DSLs.

1.3 Outline of the Dissertation

This dissertation is organized as follows:

Part I (Chapter 1) provided an overview of the dissertation as well as an introduction to partial evaluation. We first reviewed the basic concept of partial evaluation, its advantages and disadvantages for compilation application. To distinguish PE from traditional optimization techniques, we compared it to constant propagation as an example. The brief survey of partial evaluation in Section 1.2 introduced the origin of PE, its early applications, classification and the state of the art. We briefly explained our thesis statement that specialized PE is ideal for DSL compilation but left the details to following chapters.

Part II (Chapter 2) introduces the other important concept in this dissertation: domain-specific languages. The purpose of this part is to provide readers with some background knowledge of domain-specific languages and the compilation challenges they pose. We answer three questions here: why DSLs are useful in their domains, why partial evaluation can be effective for DSLs and why we must use specialized PE techniques instead of general ones to solve the problems of DSL compilation.

The answer to these questions motivates our consideration of DSLs in our research: a DSL, whose syntax is usually simpler than a GPL, relieves the compiler from complex semantic analyses but requires the partial evaluator resident in the compiler to understand its specific model deeply to achieve effective optimization results.

Part III (Chapter 3–Chapter 5) is our main technical contributions. We present three PE techniques for code generation that are applied to two concurrent, deterministic DSLs. These techniques remove concurrency (Chapter 3), enable separate compilation (Chapter 4) and unroll recursion (Chapter 5). Although the first two are for Esterel, a concurrent synchronous language, they approach the language differently. The specialization processes, therefore, are also different.

The algorithm described in Chapter 3 enables the efficient simulation of synchronous concurrent programs on a single-threaded processor. Synchronous languages, such as Esterel, provide embedded system designers a convenient tool that guarantees deterministic concurrency. However, it is nontrivial to statically schedule such a concurrent program running on a single-threaded processor. The data dependence among threads may cause frequent switches that bring considerable overhead. The solution we propose is to first eliminate as many control dependences in the program as possible, i.e., to break the source code into many small and concurrent pieces. The newly exposed concurrency, instead of introducing higher scheduling overhead as one would imagine, in fact provides the scheduler more choices and enables it to form larger and hence fewer atomic blocks. In this way, it minimizes switching overhead and generates much more efficient code.

The second algorithm we implemented for Esterel enables separately compiling code

segments of a synchronous design (Chapter 4). For a large design, it is normal to first code and test every module individually, then assemble them. However, if some modules form a communication cycle, the inputs that rely on other modules' outputs may not be available at run time to begin with. It would be very complex to write a program that handles all these cases explicitly. In this case, partial evaluation provides us an automatic way to infer the extra behavior. Performing an abstract three-valued simulation, the compiler generates descriptions that respond to unknown inputs. To keep the size of the generated code under control, we try to identify equivalent states during the simulation. The generated code, therefore, can tolerate unknown inputs and be compiled separately.

In Chapter 5, we present a PE technique that statically elaborates recursive function calls in SHIM programs. For a valid hardware design, it produces non-recursive code that is guaranteed to use bounded resources. Like Esterel, SHIM provides deterministic concurrency but presents it in an asynchronous model that uses rendezvous-style communication. Its recursive function calls enable users to construct concurrent structures dynamically. To make such programs predictable yet flexible, we use partial evaluation to eliminate recursion and replace it with static concurrent structures when possible. The algorithm applies customized constant propagation and function inlining to unroll cycles in the function call graph (recursive calls) as well as those in the control-flow graph (loops).

These various customized PE techniques illustrate the potential of partial evaluation to effectively solve compilation challenges of DSLs. They also show that aggressive optimization of a DSL requires a PE technique to be designed carefully to fit the specific model defined by the language's semantics.

Chapter 2

Domain Specific Languages

In this chapter, we introduce some domain-specific languages to illustrate the importance of DSLs and the challenges they present for compiler construction. Understanding these issues will help the reader to better appreciate the major technical contributions in the later chapters. We start with a special category, deterministic concurrent languages (Section 2.1), whose model has been used in hardware design for decades and which has also been gradually adapted to software design. Most of our research is based on these kinds of DSLs, especially on Esterel and SHIM, whose interesting models and challenging features inspired our work. We compare these two DSLs to Bluespec, another concurrent language for hardware design, to illustrate the different compilation challenges even for languages in the same application domain.

In Section 2.2, we present AG, a DSL of our own design. We include it to emphasize the importance and unique aspects (challenges) of DSLs, and to demonstrate how a DSL can facilitate the design of special-purpose systems (e.g., hardware circuit design, dataflow analysis, etc.). AG is designed to generate dataflow analyzers. We focused on making its syntax concise with an affordable performance penalty. At the end of Section 2.2, we introduce some related work and conclude.

Besides the languages' diversity, the comparison among the hardware design languages illustrates one of our hypotheses: specialized partial evaluation techniques rather than gen-

eral ones are needed to achieve aggressive optimization for DSLs. Therefore, a deep understanding of a DSL's semantics is essential for developing effective PE techniques for it.

A domain-specific language is designed to serve a specific field of programming, such as Verilog for RTL (Register-Transfer Level) design, Matlab for math computation, HTML for web page description, etc. They are usually designed with some specific syntax constructs and combined with some built-in facilities that make the design work more convenient and efficient.

The diversity and sophistication of the engineering industry inspired the birth of DSLs. In addition, a well-developed DSL may even boost the prosperity of the related domain. Verilog [71], for example, enabled the automation of circuit design, which heavily depended on manual design before 1980s. Manual design was extremely time-consuming and tedious; every chip at that time could only contain hundreds of transistors. During 1980s, things changed with the automation of circuit design, which enabled the faster design of larger systems. Many languages and tools were developed specifically in this area at the same time. Verilog, which started as a simulation language to describe the algebra of digital logic computation [38], soon became popular both because of its flexible syntax for describing test benches and because of its integrated high-performance simulator. Later, Synopsys adopted Verilog for RTL logic synthesis. It has been very successful since then. Verilog helped users to better understand the behavior-level modeling of the circuit, which in fact sped up the automation process of hardware design. Now we are able to put millions of transistors on a chip, which would not have happened without the invention of Verilog. Verilog shows how a well-designed DSL can have a major beneficial impact on a discipline — in this case, hardware engineering.

Compared to a general-purpose language (GPL), a DSL usually has a simpler syntax designed to concisely fit the logic and behavioral model in its applied field. Table 2.1 summarizes the differences between DSLs and GPLs. For example, when modeling a control-dominated embedded system, which has to meet both hard time control and ef-

	DSLs	**GPLs**
concise syntax	very	yes
general application	no	yes
specific model	yes	no
difficulty of semantic analysis	low	high
effective PE technique	specialized	general

Table 2.1: Comparing DSLs and GPLs

ficiency requirements, Esterel can provide a deterministic and concurrent solution that is more elegant and efficient than a C solution. GPLs are not an ideal solution in this specific domain because the designers' concerns are not well addressed by any GPL. Consider Java. Garbage collection, which has been welcomed by general users, turns out to be the reason that embedded system designers reject the language; losing control of memory makes them worry about unpredictable behavior occurring in a system.

The characteristics of DSLs listed in Table 2.1 relieve compilers from complex semantic analysis but place higher optimization requirements on these compilers. As we demonstrate in the next few chapters, partial evaluation does a good job of optimization. It can effectively improve the performance of the generated code. However, the various models of DSLs demand that specific partial evaluation techniques have to be designed for different models to achieve the best optimization result. The partial evaluation technique we introduce in Chapter 3, for example, works well on Esterel, but would not be helpful for VHDL. By analyzing the data and the control dependencies in Esterel programs, the compiler increases the degree of concurrency in the code and thus provides more flexibility to the optimization step that manages to generate efficient code. General PE techniques would not work well in this case since Esterel is a concurrent language and the primary performance penalty comes from context switching between threads. So we customized a graph transformation that was originally used for sequential program optimization rather than using general PE techniques. Moreover, we specialized the data dependency analysis

for the synchronous communication model in Esterel. Our experimental results show a more than four-fold speedup over existing tools, much better than what general PE could achieve.

By comparing and analyzing different models, this chapter explains why the partial evaluation techniques described in following chapters need to be radically different.

2.1 Deterministic Concurrent Languages

From multithreaded programming to multi-core system design, concurrency has many applications. But many concurrent models do not guarantee determinism, i.e., a program may behave differently at different times, even for the same input. To address this concern, certain DSLs for hardware design define strict semantics to ensure both concurrency and determinism. Esterel, for example, allows concurrent threads to communicate through signals in a single clock cycle but in a strict manner, i.e., all readers must wait for other writers that set the signal's value. On the other hand, SHIM, which does not provide global variables, instead uses single-input channels for interprocess communication. Through these strict semantics, these DSLs provide ideal solutions for designs that demand determinism and high efficiency. Embedded system design, which is naturally described as a combination of concurrent processes and does not expect any indeterminate behavior, is a good example of where we need these deterministic concurrent DSLs.

2.1.1 Esterel

Designed by Berry in 1982, Esterel [11] is a synchronous, cycle-accurate parallel language that uses a strict communication pattern. Dedicated to reactive systems, it emphasizes that users can describe control concisely. An Esterel program consists of several modules, each of which has inputs and outputs. A module, as shown in Figure 2.1, may contain multiple threads, separated by double bars. They march in step to a global clock and communicate with each other using a disciplined mechanism. In each clock cycle, the

```
module example:

input A;
output B, C, D;

  loop  % Thread I
    present B then
      emit C
    end present;
    pause
  end
||
  loop  % Thread II
    present T else
      present A then
        emit B
      else
        present C else
          emit D
        end present;
      end present;
    end present;
    pause
  end

end module
```

Figure 2.1: An example in Esterel

program computes its outputs and progresses to the next state based on its inputs and the previous state.

Signals carry and deliver the major information throughout the program by broadcasting. These signals can be classified into input, output and local signals. A module's interface, for example, is composed of input/output signals. Normally, a pure signal in Esterel carries either a present or an absent Boolean value; such a value for an input signal is determined by the environment. An output or local signal's value, by contrast, can be set by a statement. This value does not persist between cycles. Besides the values of present and absent, we will introduce the third possible value of a signal - unknown - in Chapter 4, which

may occur when some input signals are intermediate outputs of other running modules.

The syntax of Esterel is not complicated. We show a small part of it in the example in Figure 2.1, which implements two synchronized threads. Every thread in the example is enclosed in an iterative *loop* statement that takes more than one cycle to complete because of the *pause* statement, which idles for a cycle. Esterel does not allow intra-cycle loops. The other statements, *present* and *emit*, execute within a cycle. The *present* keyword tests a given signal's value and *emit* sets a signal to present, respectively. Comments are denoted by %.

Despite its concise semantics, Esterel can be used to define quite complicated interactions between threads, which can make it hard to generate efficient code. To simulate concurrent threads on a single-threaded processor, threads may have to interleave during execution. Instead of shared memory or semaphores, threads in Esterel communicate through signals and follow the rule of reader-after-writer to ensure determinism. In Figure 2.1, for example, Thread I has to start running only after Thread II since Thread II may set signal *B*'s value, which Thread I requests. However, the else branch of Thread II contains a present test on signal *C*. To acquire *C*'s value, the program is forced to switch from Thread II to I. Such sequential ordering is not always so straightforward and may involve many possible choices during compilation. Therefore, minimizing the number of context switches becomes the key to boosting the performance of generated code.

The synchronous and imperative semantics of Esterel simplify programming but complicate code generation. Esterel requires any implementation to deal with three issues: the concurrent execution of sequential threads of control within a cycle, scheduling constraints among these threads due to communication dependencies, and how (control) state is updated between cycles. To solve these issues, many different techniques have been proposed. The Esterel V3 compiler translated programs to automata. This produces quite efficient code but the size of the generated code may be exponential. V5 avoided the scale problem by generating circuit-like code, but this turned out to be slow at run time. Later, Potop-Butucaru [63] created a new intermediate representation (IR) and optimizations that

greatly improved the generated code. Based on this new IR and the work by Edwards [29], we developed a compiler that, by analyzing data and control dependencies in a program, makes aggressive optimizations and generates efficient and compact code.

2.1.2 SHIM

SHIM [32], invented by Edwards & Tardieu recently, provides a solution for designing heterogeneous embedded systems: systems combined of software and hardware. Software modules are commonly event-driven, flexible and fit well with the asynchronous model. SHIM takes the asynchronous approach where threads only synchronize with others when they must communicate.

In heterogeneous systems, especially when hardware and software have to communicate frequently, the SHIM model shows advantages over synchronous languages. The Robby Roto game, a traditional video game system, is a good example. The software typically runs at a frequency as low as 180 Hz, while the the hardware's frequency is 14 MHz - about 80,000 times faster. Obviously it would be more efficient to simulate these two at different clock frequencies and design an asynchronous interface for them. Flexibility is beneficial in these kinds of circumstances.

The syntax of SHIM is similar to C, but its use of concurrency and its rendezvous communication facility make it different. Figure 2.2 shows a sample program that deals with a sequence of integers. In this program, there's only one function main() that contains three threads where Thread II and III respectively receive even and odd numbers from Thread I. These threads are running concurrently, as the keyword *par* defines, and synchronize on channels *odd* and *n*. The *send* and *recv* statements indicate where a thread rendezvous on a channel. Unlike when a thread communicates through variables, a thread will block on a channel if its peer — another thread that communicates on the channel — is not ready. The *next* keyword represents *recv* when appears on the right side of an assignment and *send* when it appears on the left. In this program, *next odd* in Thread I means *send* since it sets *odd*'s value in the statement. The other two threads receive *odd*'s value. A channel

```
void main() {
  int n = 0;
  bool odd = 1;

  try {
    {               //Thread I
      for (;;){
        if (n < 10){
          next odd = 1 - odd;
          send n;
          n = n + 1;
        }
        else
          throw T;
      }
    }
    par
    {               //Thread II
      for (;;)
        if (!(next odd))
          recv n;
    }
    par
    {               //Thread III
      for (;;)
        if (next odd)
          recv n;
    }
  } catch(T) {}
}
```

Figure 2.2: A simple example in SHIM

must have exactly one sender but may have multiple receivers. This scheme makes the communication among concurrently running processes an arbitrary graph, which may be cyclic.

SHIM defines a complicated exception handling system that cooperates with concurrency. Generally speaking, a thread that communicates with any other thread will "die" if its peers are aborted. For example, Thread II (Figure 2.2), which contains an infinite *for* loop, seems to run forever. Nevertheless, when its peer Thread I throws an exception *T* and terminates, Thread II will terminate as well, as will Thread III. This process is also called "poisoning" because it appears that the termination due to the exception is being propagated among threads through channels.

The computational model of SHIM is based on Kahn networks [49]. Although the scheduling-independent character of the model allows the compiler to make many scheduling choices, it is still hard to generate fast code from a SHIM program. One way to make the code run fast is to know at compile time when threads will rendezvous. In the example in Figure 2.2, Thread II will only retrieve the value from Thread I through channel *n* when the next integer is even. The process of actually determining when the threads will rendezvous, however, can become fairly involved. In addition, fanout and cyclic communication may further complicate the situation; because of these challenges, a compiler may have a difficult time statically scheduling the code. Furthermore, exception handling presents another challenge. Consider three threads A, B, and C (C contains both A and B: in other words, both A and B are child threads of C). Even if thread A raises an exception, thread C can potentially keep running because B may be unaffected by the exception from A. Tardieu and Edwards provided some rules to handle exceptions in their work [70].

Recursion in SHIM presents another challenge. SHIM provides recursive function calls to enable succinct designs. However, a design with recursive calls may need unbounded resources. This is unacceptable for hardware implementations, so it is sometimes necessary during the compilation process to eliminate recursion. We propose an algorithm to transform a program with recursion to one only requiring bounded resources when possible.

```
(* descending_urgency = ''proc2, proc1, proc0'' *)

rule proc0 (cond0);
  x <= x + 1;
endrule

rule proc1 (cond1);
  y <= x;
endrule

rule proc2 (cond2);
  x <= x -1;
endrule
```

Figure 2.3: An example in Bluespec

More details are included in Chapter 5.

Although both SHIM and Esterel are concurrent, deterministic and modular, they address different concerns in embedded system designs. Esterel uses the synchronous model to provide precise control over system timing while SHIM uses the asynchronous model for flexibility.

2.1.3 Bluespec

Bluespec is another concurrent and deterministic tool for hardware design. It is based on Hoe and Arvind's 1999 proposal [44] of a synthesizable Term Rewriting Systems (TRS) model for microprocessor designs. The tool provides a lot of facilities, including optimizations, long-bit-vector support and verification. These facilities are attractive for high-level designs.

The language is called Bluespec Verilog (BSV) since its syntax is similar to Verilog but without the *always* keyword. Like other hardware design languages, it is strongly-typed and side-effect free. A BSV design is composed of modules. Each module includes a description of the system state elements, such as registers and atomic behavioral components (rules). The segment of BSV code in Figure 2.3 contains three rules and registers *x* and *y*.

Each rule's body describes the allowable sequence of state transitions and its head specifies the condition under which the rule is enabled. The rule *proc0*, for example, increases *x*'s value when the Boolean condition *cond0* is satisfied. Although rules can involve complicated transitions, such as ones containing method calls, every rule is atomic, i.e., the specified state transition cannot be interrupted.

The main challenge for the BSV compiler is to build a run-time schedule that assures the atomicity constraints of the rules. Unlike Esterel and SHIM where a designer defines the control logic of the system, Bluespec has the compiler infer the optimal control structure from the rules. In other words, the TRS model implies what sequences of state transition are possible. Rules that do not modify the same state element may be scheduled to run concurrently. For example, the rules *proc0* and *proc1* have no conflict, so they can run in parallel if their conditions are true simultaneously. The rules *proc0* and *proc2*, on the other hand, both write to register *x*. When they are both enabled, the compiler has to check the priority specification, such as the first line in the example (Figure 2.3), or raise an error. Since a rule can invoke method calls, the state it reads or writes may be distributed across several modules. This makes the automatic generation of control logic even harder. Furthermore, any change in a module or its related modules must be verified to guarantee atomicity.

A smart schedule may lead the compiler to generate code as efficient as hand-coded Verilog [4], but heavy reliance on the compiler may raise concerns of losing control over the behavior of the system, especially for large designs.

2.2   A Little Language for Generating Dataflow Analyzers

To illustrate in detail how a domain-specific language can help to simplify the development process, we present a little language called Analyzer Generator in this section. The concise syntax of the language can greatly reduce code size. We also introduce some potential PE optimization opportunities provided by the language at the end of this section.

Dataflow analysis is a well-understood and very powerful technique for analyzing programs as part of the compilation process. Virtually all compilers use some sort of dataflow analysis as part of their optimization phase. However, despite being well-understood theoretically, such analyses are often difficult to code, making it difficult to quickly experiment with variants.

Our domain-specific language, Analyzer Generator (AG), synthesizes dataflow analysis phases for Microsoft's Phoenix compiler framework. AG hides the fussy details needed to make analyses modular, yet generates code that is as efficient as the hand-coded equivalent. One key construct we introduce allows IR object classes to be extended without recompiling.

Through AG, we demonstrate how necessary and helpful a DSL is when we design programs applied to a specific domain. Experimental results on three analyses show that AG code can be one-tenth the size of the equivalent handwritten C++ code with no loss of performance. It shows that AG can make developing new dataflow analyses much easier.

2.2.1 Coding Dataflow Analysis Algorithms

Modern optimizing compilers are sprawling beasts. GCC 4.0.2, for example, tips the scales at over a million lines of code. Much of its heft is due simply to its many features: complete support for a real-world language, a hundred or more optimization algorithms, and countless back-ends. But the intrinsic complexity of its internal structures' APIs and the verbosity of its implementation language are also significant contributors.

We address the latter problem by providing a domain-specific language, AG for "Analyzer Generator," for writing dataflow analysis phases in Microsoft's Phoenix compiler framework. Experimentally, we show functionally equivalent analyses coded in AG can be less than one-tenth the number of lines of their hand-coded C++ counterparts and have comparable performance.

Reducing the number of lines of code needed to describe a particular analysis can reduce both coding and debugging time. We expect our language will make it possible to

quickly conduct experiments that compare the effectiveness of various analyses. Finally, by providing a concise language that allows analyses to be coded in a pseudo-code-like notation mimicking standard texts [1], compiler students will be able to more quickly code and experiment with such algorithms.

One contribution of our work is a mechanism for dynamically extending existing classes. In writing a dataflow analysis, it is typical to want to add new fields and methods to existing classes in the intermediate representation (IR) in the analysis. Such fields, however, are unneeded after the analysis is completed, so we would like to discard them. While inheritance makes it easy to create new classes, most object-oriented languages do not allow existing classes to be changed. The main difference is that we want existing code to generate objects from the new class, which it would not otherwise do.

The challenge of extending classes is an active area of research in the aspect-oriented programming community [52], but their solutions differ from ours. For example, the very successful AspectJ [51] language provides the intertype declarations that can add fields and methods to existing classes. Like ours, this technique allows new class fields and methods to be defined outside the main file for the class, it is a compile-time mechanism that actually changes the underlying class representation, requiring the original class and everything that depends on it to be recompiled. In AG, only the code that extends the class must be recompiled when new fields are added.

MultiJava [22] provides a mechanism that is able to extend existing classes without recompiling them, much like our own, but their mechanism only allows adding methods, not fields, to existing classes.

In AG, we provide a seamless mechanism for adding annotations to existing IR classes. In AG code, the user may access such added fields with the same simple syntax as for fields in the original class. Adding such fields does not require recompiling any code that uses the original classes.

We implemented our AG compiler on top of Microsoft's Phoenix, a framework for building compilers and tools for program analysis, optimization, and testing. Like the SUIF

system [76], Phoenix was specifically designed to be extensible and provides the ability, for example, to attach new fields to core data types without having to recompile the core. Unfortunately, implementing such a facility in C++ (in which Phoenix is coded) has a cost both in the complexity of code that makes use of such a facility and in its execution speed. Experimentally, we find the execution speed penalty is less than factor of four and could be improved; unfortunately, the verbosity penalty of using such a facility in C++ appears to be about a factor of six. Reducing this is one of the main advantages of AG.

2.2.2 The Design of AG

AG is a high-level language that provides abstractions to describe iterative dataflow analyses. The AG compiler translates an AG program into C++ source and header files, which are then compiled to produce a Dynamically Linked Library (DLL) file. (Figure 2.4) This DLL can then be plugged in to the Phoenix compiler and invoked just after a program is translated into Phoenix's Middle Intermediate Representation (MIR).

Our generated plug-in extends IR objects to collect information and invokes a traversal that is part of the Phoenix framework to perform iterative analysis. This traversal function invokes computations defined in the AG program.

We follow the classical dataflow analysis approach. An AG program implicitly traverses the control-flow graph of the program and considers a basic block at a time. Inside each block, the analysis manipulates its constituent instructions and operands. We thus chose to make blocks, instructions, and operands basic objects in AG. Phoenix, naturally, already has such data types, but AG makes them easier to uses since our language has a deeper understanding of them.

One of the main contributions of AG is the ability to add attributes and computations to these fundamental data types. This facility relies on mechanisms already built into Phoenix, but because of the limitations of C++, making use of such mechanisms is awkward and tedious to code. AG makes it much easier.

To simplify the description of computation functions, we included new statements in

DataflowAlgorithm.ag

AG Translator

DataflowAlgorithm.Phx.cpp/.h

AG Library Files → Phx C++ Compiler ← Phx Library Files

DataflowAlgorithm.dll
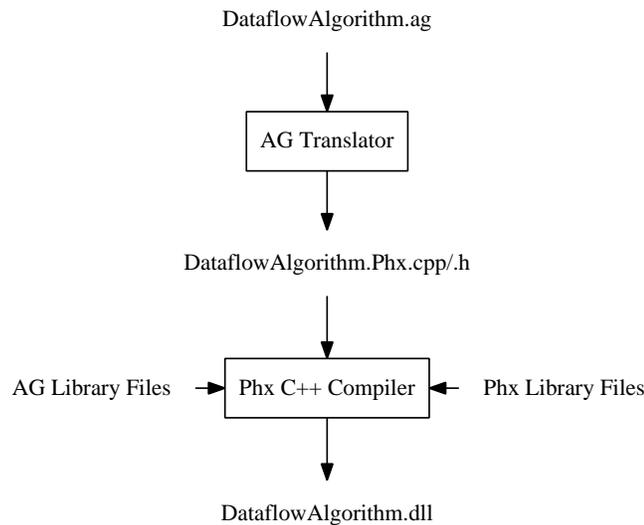
Figure 2.4: The operation of the AG framework

AG such as *foreach* and data-flow equations like those found in any compiler text. We also introduced a *set* data type since data collected during dataflow analysis usually takes the form of sets.

AG relies on the Phoenix Traverser class. This is an iterative traverser that does not guarantee boundedness. See Nielson and Nielson [60] for a discussion of the issues in guaranteeing boundedness.

2.2.3 Program Structure and Syntax

The AG language is designed for dataflow analysis. It provides abstractions for the common features of iterative intraprocedural analysis. For user convenience and adaptability, we chose a syntax similar to that of C++ and added a variety of new statements and constructs.

Figure 2.5 shows the structure of a typical AG program to describe an analyzer. It defines a new, named phase, extends a number of built-in Phoenix classes with new fields and methods to define what information to collect, and finally defines a transfer function for the dataflow analysis.

```
Phase name {
    extend class name {
        field declarations...
        method declarations...
        void Init() { ... }
    }
    ⋮
    type TransFunc(direction) {
        Compose(N) { ... }
        Meet(P) { ... }
        Result(N) { ... }
    }
}
```

Figure 2.5: The structure of an AG program

An *extend class* defines a new ɪʀ class that uses the Phoenix dynamically extensible ɪʀ class system. New fields and methods declared in an extend class are added as new class members. The user may directly refer to them as if they were members of the original class (our compiler identifies such fields and generates the appropriate Phoenix code to access and call members of such extended classes). Notice the methods declared in an extend class are "private," i.e., they can only be applied to the corresponding extend object, or in other methods declared under the same extend class. Currently, we only support extending Block, Instr, and Opnd classes.

In each extend class, the Init method behaves (and is executed as) an initializer just after the constructor for the extended class.

Each phase has a single TransFunc that defines the return type and iteration direction

(backward or forward) of the analyzer and, more importantly, the equations applied during the analysis. The body of a TransFunc may define functions, especially three reserved functions: Compose, Meet, and Result. Compose and Meet functions are applied when the traverser visits every blocks. The Compose function defines the computation inside a block using global data. The Meet function defines the computation performed between blocks, i.e., to merge data from the exit of the predecessor to the entry of the successor. The Result function defines operations to be performed just after the iteration. It usually propagates information to the objects that make up the blocks, such as instructions. Other functions may be declared in the TransFunc; they can be called by the three reserved functions or each other.

The user may embed arbitrary C++ code in the body of these methods. Such code segments are transparent to AG compiler, which simply includes them verbatim in the generated code.

We derived the syntax of AG from C++. We present its complete syntax in the appendix; Table 2.2 provides a summary. Below, we provide some details about its design.

Set is a data type similar to *set* in the C++ standard library. It can only apply to the reserved classes and actually refers to a set of IDs. For example, "Set<Instr>" will be translated into a bit-vector mapped on IDs of instructions in implementation. The *Map* type is similar.

During the analysis, the most relevant data are those with information for the entry and exit points of each block, so we introduced the *In* and *Out* data set as built-in variables.

Except for the two logical operators, the operators in Table 2.2 can be applied both to integers and *Set*-valued variables. Using the +, −, and * operators generate code that perform Or, Minus, and And operations on bit vectors.

In dataflow analysis, one often needs to iterate over a subset of objects, so we added a *foreach* statement to do this. *Foreach* is a predicated iterator, meaning that it steps through the members of a set and performs actions on only selected members of the set. The user does not have to declare an iterator specifically, just a variable of the type over which the

data types	Set Map int bool void
special variables	In Out
operators	+ − ∗ = += −= ∗= && ‖
built-in classes	Opnd Instr Block Alias Expr Func Region
special methods	Init Compose Meet Result
built-in functions	DstAliasTable SrcAliasTable Print
built-in constants	Forward Backward
declarations	Phase *identifier* (*parameter list*) { ... }
	extend class *type* { ... }
	type TransFunc (*direction*) { ... }
statements	*lvalue* = *expression*;
	if (*expression*) { ... } else { ... }
	/% *arbitrary C++ code* %/
	foreach (*type var* in *range* where *cond. direction*) { ... }
	phoenix-iterator (...) { ... }

Table 2.2: AG Syntax Summary

iteration is occurring and the set on which to iterate. The user may also specify a condition that acts as a filter and a direction (Forward/increase or Backward/decrease). The condition is described with the *where* keyword. The syntax is shown in Table 2.2.

The *type*, *range* and *condition* allowed are listed in the attached syntax table. The "where *condition*" and "*direction*" parameters are optional.

Such *foreach* statements are translated to conditional for loops in the C++ and use the iterator macros in the Phoenix framework. Note that the *foreach* statement, especially the predication, is not strictly necessary (an additional *if* is sufficient), but the same can be said of C's *for* statement.

If the *range* is a Set, the *type* must match its content. Otherwise, if the *range* is a class, the *type* must match one of its members. For example, each instruction contains a list of

operands, so we can specify a *type* of Opnd and a *range* of an instruction. Also, the user may specify a *condition* of "dataflow && dst" to iterate over dataflow-related destination operands in the list.

Phoenix provides a number of iterator macros, which can be used in AG almost verbatim (see Figure 2.6 Line 9). The only difference is that in C++, a matching "next" macro must follow the use of each iterator macro (see Figure 2.7 Line 14); this is not necessary in AG.

DstAliasTable is a reserved function that takes an alias tag *x* as parameter and returns a set of destination operands whose alias-tag is *x*. Similarly, *SrcAliasTable* returns all source operands with the same alias-tag.

2.2.4   An Example

To illustrate AG, we present a complete example: the classical "reaching definitions" analysis. The complete AG source is in Figure 2.6.

This algorithm computes the sets of definitions that reach the entry and exit points of each basic block in a program. Following the Dragon book [1], a definition of a variable is the operand in an instruction that may assign to the variable. In the Phoenix ir, each instruction has source operands and destination operands. For reaching definitions, we are concerned mostly with the destinations.

The whole analysis is defined as a phase called *ReachingDefs* (line 1 of Figure 2.6). The rest of the analysis consists of extend classes that add fields and computations to the built-in data types for operands, instructions, and basic blocks, and description of transfer functions.

Extend classes augment existing data types with additional fields in which to collect information and procedures for collecting it. This is similar to extending a base class in an object-oriented language, but differs because the new attributes are actually attached to objects of the "base class" itself at the language level, not just in objects of derived classes (the C++ code we generate from AG actually uses class inheritance). But a user can refer

```
1  Phase ReachingDefs {
2    extend class Opnd {
3      Set<Opnd> Gen;
4      Set<Opnd> Kill;
5      void Init() {
6        Opnd opnd = this;
7        if (opnd->IsDef) {
8          opnd->Gen += opnd;
9          foreach_must_total_alias_of_tag(alias_tag, opnd->AliasTag, AliasInfo)
10           opnd->Kill += DstAliasTable(alias_tag);
11         opnd->Kill -= opnd; }
12     }
13   }
14
15   extend class Instr {
16     Set<Opnd> Gen;
17     Set<Opnd> Kill;
18     void Init() {
19       Instr instr = this;
20       foreach (Opnd dstOpnd in instr where (dataflow && dst)) {
21         instr->Gen += dstOpnd->Gen;
22         instr->Kill += dstOpnd->Kill; }
23     }
24   }
25
26   extend class Block {
27     Set<Opnd> Gen;
28     Set<Opnd> Kill;
29     void Init() {
30       Block block = this;
31       foreach (Instr instr in block) {
32         block->Gen = instr->Gen + (block->Gen - instr->Kill);
33         block->Kill = block->Kill + instr->Kill - instr->Gen; }
34     }
35   }
36
37   Set<Opnd> TransFunc(Forward) {
38     Compose(N) { Out = In - N->Kill + N->Gen; }
39     Meet(P) { In += P->Out; }
40   }
41 }
```

Figure 2.6: A Complete AG analysis: Reaching Definitions

```
1 class OpndExtensionObject : public Phx::RbagGenTest::AG::OpndExtensionObject {
2   PHX_DECLARE_PROPERTY(Phx::BitVector::Sparse *, Gen);
3   __PHX_DEFINED_VIRTUAL_GET_PROPERTY(Phx::BitVector::Sparse *, Gen) __const;
4   __PHX_DEFINED_VIRTUAL_SET_PROPERTY(Phx::BitVector::Sparse *, Gen);
5   Phx::BitVector::Sparse * _local_Gen;
6 }
7 void OpndExtensionObject::Init( Phx::FuncUnit *func_unit,
8                                 Phx::BitVector::Sparse *PHX_ARRAY(dst_alias_table)) {
9   Phx::IR::Opnd *opnd = _this;
10  if(opnd->IsDef) {
11    this->Gen->SetBit(this->uid);
12    foreach_must_total_alias_of_tag(alias_tag, opnd->AliasTag, func_unit->AliasInfo)
13      this->Kill->Or(dst_alias_table(alias_tag));
14    next_must_total_alias_of_tag;
15    this->Kill->ClearBit(this->uid);
16  }
17 }
18 void IterateData::Merge(Phx::DataFlow::Data *dependent_block_data,
19   Phx::DataFlow::Data *effected_block_data, Phx::DataFlow::MergeFlags flags) {
20   IterateData * dep_block_data = PTR_CAST(IterateData *, dependent_block_data);
21   Phx::BitVector::Sparse * Out = dep_block_data->Out;
22   if(flags & Phx::DataFlow::MergeFlags::First) In = Out->Copy(); else In->Or(Out);
23   dep_block_data->Out = Out;
24 }
25 void Traverser::InitData(Phx::BitVector::Sparse *PHX_ARRAY(dst_alias_table)) {
26   foreach_block_in_func(block, funcUnit) {
27     foreach_instr_in_block(instr, block) {
28       foreach_dataflow_dst_opnd(dstopnd, instr) {
29         OpndExtensionObject *ext_dstopnd = OpndExtensionObject::GetExtensionObject(dstopnd);
30         ext_dstopnd->Init(funcUnit, dst_alias_table);
31       } next_dataflow_dst_opnd;
32       InstrExtensionObject *ext_instr = InstrExtensionObject::GetExtensionObject(instr);
33       ext_instr->Init(funcUnit->Lifetime);
34     } next_instr_in_block;
35     BlockExtensionObject *ext_block = BlockExtensionObject::GetExtensionObject(block);
36     ext_block->Init(funcUnit->Lifetime);
37   } next_block_in_func;
38 }
```

Figure 2.7: Part of the Phoenix (C++) code generated by the AG compiler for the reaching definitions example

to new attributes as if they were already in the original class. Consider the Opnd extend class (lines 2–13). This adds two attributes to each operand, operand sets named *Gen* and *Kill*. As usual, the *Gen* set contains operands that are defined within the block and available immediately after it in the source code.

The *Init* function initializes the values of the *Gen* and *Kill* fields. The two sets are implemented as bit vectors—see Lines 2–5 in Figure 2.7 for the declaration of *Gen*; Lines 7–17 show the translation of the *Init* function. The body of *Init* adds destination operands to the *Gen* set. Similarly, all other destination operands in the built-in destination-opnd-map-to-alias-tag table (DstAliasTable) that have the same alias tag as the operand (i.e., when both modify the same memory location) are added to the *Kill* set (Lines 7–11).

The *Instr* and *Block* extend classes add *Gen* and *Kill* sets to each of their classes and populate these sets with data from *Opnd* and *Instr* objects respectively. Lines 26–37 in Figure 2.7 call the three *Init* functions (the translation of the other two are not shown). Note that this function is synthesized completely from how this data is used in the analyzer, not from explicit code in the AG source.

After collecting *Gen* and *Kill* sets for blocks, the algorithm specifies some details of the main analysis iteration. At the beginning of the transfer function *TransFunc*, the iteration is declared to proceed in the forward direction and return a set of *Opnd* objects.

The extend classes are based on original IR classes. The example in Figure 2.6 shows that, to refer to fields from the extend class (e.g., Figure 2.6, Line 8, "opnd->Gen"), the user may use the same notation as for those in the base class (e.g., Figure 2.6, Line 9: "opnd->AliasTag"). These two references generate very different C++ code (c.f. Figure 2.7, Lines 11 and 12).

As usual, we assume there are unique entry and exit points in the control flow graph for each block. "In" and "Out" are two built-in data sets related to the entry and the exit points respectively. The definition for TransFunc head declares the type of "In" and "Out" sets as holding operands. These two sets are usually used in the transfer function to pass data.

Compose and Meet are the two main functions for defining the transfer function. In

this program, they specify the two groups of dataflow equations in the standard way [1, Eq. 10.9]:

$$in[B_i] = \bigcup_{B_j \text{ a predecessor of } B_i} out[B_j]$$

$$out[B_i] = gen[B_i] \cup (in[B_i] - kill[B_i]).$$

The first equation is exactly and simply included in the Meet function (Line 39), which computes the effect of the exit-point data from predecessors to the entry-point data of the current block in the iteration. *In* is related to the current block being visited, while *Out* is related to the block *P* that is passed to the *Meet* function. By default, the argument for the *Meet* function is a basic block that represents an arbitrary predecessor of the current block. As shown in Figure 2.7 lines 18–24, the data equation is translated into bit-vector manipulations.

The second dataflow equation is included in the *Compose* function (Line 38), which computes the data transformation globally from the entry point to the exit point for a single block. Declared as an argument to the *Compose* function, variable *N* is an extended object of the block by default. Since *Gen* and *Kill* are fields that have been added to the Block class (lines 27 and 28), they can be referred to as members of *N*.

A complete AG program is translated into a C++ program that is compiled as a plug-in phase that can be invoked as part of the Phoenix compilation processes. It initializes all extended objects first, then executes the forward traverser, which applies the dataflow equations to iteratively compute on the blocks following the structure of the control-flow graph until the *In* sets converge for every block. The generated code uses the machinery built into the Phoenix framework to do this; an AG user does not write code for this.

2.2.5 Experimental Results

We tested AG on three analyses: reaching definitions, live variables, and uninitialized variables. We chose these three examples because a hand-written version of each, done by experienced programmers, already existed in Phoenix. We compared the size and speed of

	Reaching Definitions	Live Variables	Uninitialized Variables
C++ LOC (manual)	791	303*	108†
AG LOC (manual)	41	55	94
C++ LOC (generated)	626	519	682
C++ runtime	7.3s	0.8s	†
AG runtime	7.4s	3.1s	13.6s

Table 2.3: Experimental results: size and speed of AG-generated code vs. handwritten.
*The manually coded live variable analysis uses hard-coded fields, which makes it simpler at the expense of being far less modular.
†The manually coded uninitialized variables analysis relies on the Phoenix SSA library not included in this count. This is a very different architecture than the code generated by AG.

the generated code with the manually written version for the first two examples because, like our generated code, they use the Traverser class in Phoenix. The manually written version of uninitialized variables used Phoenix's static single-assignment code, which AG does not take advantage of, so we did not experiment with it.

Table 2.3 shows our results. "LOC" indicates the number of lines of code excluding comments; times are in seconds. We computed the average run times of these plug-ins by running compiler with the plug-in, running the compiler without the plug-in, and subtracting these two running times. The times are thus a little suspect because they also include the time to load and initialize the plug-in itself.

In each test case, the C++ code generated by the AG compiler is more than six times the size of the AG source. Even better for AG, the manually written code for reaching definitions is even larger than the generated code. That is because the AG library files include commonly used code and default methods, for example, the constructor of the phase.

The manually written live-variables code is smaller than the generated C++ code for

that analysis, but this is because the manually written code does not use the (verbose) Phoenix extend objects.

We ran the generated Phoenix C++ code on a laptop with a 2.0 GHz Pentium-M processor running Windows XP. The benchmark is the Phoenix Microsoft Intermediate Language reader, which can generate high-level intermediate representations for a variety of targets. It is about five hundred thousand lines of code.

The AG-generated code for the reaching definitions analysis runs just as fast as the manually written code on the MSIL reader. Unfortunately, the live variable analysis code runs about one-fourth as quickly, but there is a good reason for this: the manually written C++ version does not use the Phoenix object-extension facility. Instead, it simply recomputes the desired data every time it traverses a block. Thus, the speed difference here more illustrates the cost of using extension objects instead a more brute-force approach. Evidently in this example, the computation is cheap enough so that repeating it is less costly than saving and recovering it later. We include the runtime for the AG code for uninitialized variables, but do not give a time for the manually written code because it uses a completely different algorithm.

2.2.6   Related Work

The theory of dataflow analysis is well-studied. Kildall [53] was one of the first to propose a unified lattice-based framework for global program analysis. Later, Kam and Ullman [50] addressed the iterative approach and made the theory more concrete.

Wilhelm [75] notes that there are many generic theories for dataflow analysis, but few tools are built on these theories and even fewer are widely accepted. One big reason is the lack of a standard mid-level program representation. We expect the Phoenix compiler framework to address this problem, at least for object-oriented imperative languages. Another reason for the lack of tools is their complexity. Thus the focus of our work is to provide a simple language and tool for writing dataflow analyses.

Tjiang's Sharlit [72] is a tool for building iterative dataflow analyzers and optimizers.

It is built on the sUIF [76] generic compiler construction framework. However, Sharlit did not introduce a new language. It uses C++ and provides some APIs, much like the Phoenix environment, and its focus was mostly on its efficiency, not its simplicity. While it makes an implementation of an analysis much more modular, it remains difficult to use.

A few tools require an explicit definition of the lattice used in dataflow analysis. Examples include Alt and Martin's PAG [2], Venkatesh and Fischer's SPARE [74], and the flexible architecture presented by Dwyer and Clarke [28]. PAG is well-known and has been used in industry. There are many similarities between AG and PAG: both use basic blocks and unchanged-pre-condition checking to improve the speed of the generated analyzer. Both provide a "set" data type. Unlike AG, PAG requires the user to specify the lattice used during analysis, which provides more optimization choices, like widening and narrowing, and makes it easier to verify the algorithm's correctness, but this makes PAG descriptions larger and more complex.

Some tools specifically address interprocedural analysis, such as Yi and Harrison's auto-generation work [77]. We focus only on intraprocedural analysis, although many of our ideas should carry over to inter-procedural problems.

2.2.7 Conclusions

To illustrate the advantage of DSLs for programming in specific domains, we presented a DSL, AG, for writing dataflow analysis phases in Microsoft's Phoenix framework, whose succinct syntax greatly decreases the implementation's code size as well as the workload of dataflow algorithm designers. Experimental results show that manually written AG code can be less than one-tenth the size of the equivalent manually written C++ with similar performance. A key enabler for the simplicity of AG code is its mechanism for extending existing IR classes, which makes it possible to extend existing classes without recompiling them and allows user-level code to access these fields as easily as typical ones.

As a small, domain-specific language, AG has some weaknesses. Minimizing verbosity was our focus, and we did so at the loss of some flexibility. The most obvious is that the user

is forced to use the iterative analysis framework, even though Phoenix has other options, such as lattice and static single-assignment frameworks. Although AG has some high-level types such as sets and maps, its type system is limited and does not support strings, arrays, arbitrary iterators, and so forth.

AG is also currently limited to analyses running on the medium-level intermediate representation (MIR), although it could be extended to handle others. Furthermore, AG programs currently only handle user-defined variables; the many implicit temporary variables in the MIR are currently ignored. For example, the C statement on the left is dismantled as shown on the right. AG code currently ignores the temporary `t1`.

$$x = y + 3; \qquad \longrightarrow \qquad \begin{aligned} &t1 = y + 3; \\ &x = t1; \end{aligned}$$

As with many domain-specific languages, debugging AG is somewhat problematic. While we provide a print statement, AG does not have a dedicated debugger, IDE, or any of the other now-standard features in a development environment. All these could be added, but not without a fair amount of work.

Partial evaluation may be added to improve the efficiency of the generated AG program. Since AG takes an iterative framework, the dataflow analysis will terminate sooner if the relaxation process is quick. An online partial evaluation that analyzes the relaxation condition in the AG program may help. Instead of iterating on every block in an arbitrary order, the evaluator can arrange the blocks in a descending order of how fast the output converges. This should speedup the iterative analysis.

AG is constructed as a translator, so in theory most weaknesses could be fixed by extending AG, provided the new features were supported by Phoenix. It could be extended, say, to describe region-based dataflow analyses, or to describe optimizations. But it is difficult to say at what point AG would cease to be a domain-specific language and balloon into C++. This is also a general issue for DSLs.

Nevertheless, we believe that a factor of six in code-size reduction justifies the extra challenges in using a small language.

2.3 Summary

We reviewed some domain specific languages in this chapter, especially deterministic, concurrent and modular languages. In order to address a variety of concerns, DSLs, even if designed for the same problem domain, may be built on fundamentally different models. Their radically different features present particular challenges to their compilers. The comparison of Esterel, SHIM and Bluespec illustrates the differences.

Compared to a general-purpose language, a DSL usually has simpler syntax that is less flexible but more succinct and reliable for coding. Optimized for its specific model, a DSL compiler may generate very efficient code.

This chapter prepares readers with background knowledge of DSLs. The comparison between DSLs and GPLs explains that, because of the simpler syntax but special computational models of DSLs, partial evaluation may work effectively on DSLs as well. In addition, the comparison of different concurrent, deterministic DSLs illustrates that specialized PE techniques are required to solve various challenges brought by specific models.

Chapter 3

Partial Evaluation for Removing Concurrency

Generally, concurrency in embedded systems is facilitated by real-time operating systems. Such concurrency can be unpredictable and difficult to debug since the operating system does the scheduling. Synchronous concurrency on the other hand, in which a system marches in lockstep to a global clock, is conceptually easier and potentially more efficient because it can be statically scheduled. The synchronous language Esterel provides parallel constructs to define such systems. However, simulating synchronous concurrency on a single-threaded processor can be very expensive because of switching overhead between threads.

In this chapter, we introduce a partial evaluation algorithm to minimize switching overhead in Esterel. Our algorithm removes most of this overhead and generates efficient sequential code from synchronous concurrent specifications. Given a concurrent program dependence graph generated from an Esterel program, we sequentialize the concurrent code by adding a minimal amount of run-time scheduling code. With the right language and a specialized PE technique, it becomes possible to simulate concurrent programs efficiently. This work originally appeared in the proceedings of LCTES in 2004 [79].

This algorithm demonstrates that a specialized PE technique enables aggressive opti-

mization, especially for a synchronous model.

The chapter is organized as follows. We first introduce the challenge of scheduling concurrent programs, then provide a modified definition and several important properties of the program dependence graph, which is the representation used to specify computation. We then present the algorithm in detail, including the process of restructuring and code generation. Finally, we compare our results with existing techniques. We find partial evaluation can generate code that runs as much as six times faster.

3.1   Scheduling a Concurrent Program

Embedded software is often conveniently described as collections of concurrently running processes and implemented using a real-time operating system (RTOS). While the functionality provided by an RTOS is very flexible, the overhead incurred by such a general-purpose mechanism can be substantial. Furthermore, the interprocess communication mechanisms provided by most RTOSes can easily become unwieldy and easily lead to unpredictable behavior that is difficult to reproduce and hence debug. The behavior and performance of concurrent software implemented this way is difficult to guarantee.

The synchronous languages [7] provide an alternative by providing deterministic, timing-predictable concurrency through the notion of a global clock. Concurrently running threads within a synchronous program execute in lockstep, synchronized to a global, often periodic, clock.

The model of time used within the synchronous languages happens to be identical to that used in synchronous digital logic, making the synchronous languages perfect for modeling digital hardware. Hence, executing synchronous languages efficiently also improves the simulation of hardware systems.

Unfortunately, implementing such languages efficiently is not straightforward since the detailed, instruction-level synchronization is difficult to implement efficiently with an RTOS. Instead, successful techniques "compile away" the concurrency through a variety

of mechanisms ranging from building automata to statically interleaving code [31].

In this chapter, we present a technique for compiling such finely synchronized concurrent specifications that produces very efficient code. While we implemented this technique in the Columbia Esterel Compiler (CEC), our proposed algorithm starts from the well-known program dependence graph (PDG) representation [37]. In principle, then, this technique is applicable to a variety of imperative, sequential languages with concurrency.

We chose the synchronous Esterel [11] for a number of reasons. Its communication can be analyzed statically—the absence of aliasing makes it possible to statically identify all possible inter-thread communication pathways. Its control flow is acyclic and therefore easy to analyze. Also, it is a challenging language to compile because of its mix of concurrency and control flow. Existing techniques for compiling Esterel grapple with scheduling overhead, but our use of the PDG representation allows detailed instruction scheduling that effectively reduces overhead.

CEC first performs a syntax-directed translation of an Esterel program into an acyclic control-flow graph with data dependence information. It then converts this into a PDG using a slight modification of the algorithm due to Cytron et al. [27] to handle Esterel's concurrent constructs.

We present a novel algorithm that restructures a program dependence graph with arbitrary acyclic data dependencies into one that has a direct translation into sequential code. Unlike a PDG generated from purely sequential code, it it not usually possible to translate the PDG produced from Esterel directly into sequential code because communication patterns in the Esterel program may force concurrently running threads to be interleaved. This can be solved by either duplicating code, a potentially costly operation that may produce an exponential increase in code size, or by inserting additional guard variables and predicates. We take the second approach, using heuristics to choose where to cut the PDG and introduce predicates, and produce a semantically equivalent PDG that does have a simple sequential representation. We use a modified version of Simons and Ferrante's algorithm [67] to produce a sequential control-flow graph from this restructured PDG and

procedure Main

    Clear the visited set

    PriorityDFS(root node of *G*)

    Clear the schedule and visited set

    ScheduleDFS(root node of *G*)

    Restructure()

    Fuse guard variables

    Generate sequential code from *G*′

Figure 3.1: The Main procedure. The input of this procedure is a program dependence graph *G*. And the output is a segment of sequential code. *G*′ represents the graph after restructuring.

finally generate sequential C code from it.

Our algorithm works in three phases (see Figure 3.1). First, we compute a schedule—a total order of all the nodes in the PDG (Section 3.2.1). This procedure is exact in the sense that it always produces a correct result, but heuristic in the sense that it may not produce an optimal result. Second, we use this schedule to guide a procedure for restructuring the PDG that slices away parts of the PDG, moves them elsewhere, and inserts assignments and tests of guard variables to preserve the semantics of the PDG (Section 3.2.2). Finally, we use a slightly enhanced version of the sequentializing algorithm due to Simons and Ferrante to produce a control-flow graph (Section 3.2.3). Unlike Simons and Ferrante's algorithm, our sequentializing algorithm always completes because of the restructuring phase. In Section 3.3, we present experimental results showing this technique can produce code that runs as much as thirty times faster than others.

3.1.1 The Program Dependence Graph

We specify computation using a variant of Ferrante, Ottenstein and Warren's [37] program dependence graph. The PDG for a program is a directed graph whose nodes represent

statements and whose arcs represent the partial ordering among statements that must be followed to preserve the program's semantics. In some sense, the PDG removes the maximum number of dependencies among statements without changing the program's meaning.

A PDG is a rooted, directed acyclic graph $G = (S, P, F, r, c, D)$, where S, P, and F are disjoint sets of statement, predicate, and fork nodes. Together, these form the set of all nodes in the graph, $V = S \cup P \cup F$. $r \in V$ is the distinguished root node. $c : V \to V^*$ is a function that returns the vector of control successors for each node (i.e., they are ordered). Each node may have a different number of successors. Without special clarification, the term successor is referred to control successor in this section. $D \subset V \times V$ is a set of data arcs. If $c(v_1) = (v_2, v_3, v_4)$, then node v_1 can pass control to v_2, v_3, and v_4. The set of control arcs can be defined as $C = \{(m, n) : c(m) = (\ldots, n, \ldots)\}$; i.e., (m, n) is a control arc if n is some element of the vector $c(m)$. If a data arc $(m, n) \in D$, then m can pass data to node n.

The semantics of the graph relies mostly on the node types. A statement node $s \in S$ is the simplest: it represents a computation with a side effect (e.g., assigning a value to a variable) and has no outgoing control arcs. A predicate node $p \in P$ also represents a computation but has outgoing control arcs. When executed, a predicate arc passes control to exactly one of its control successors depending on the outcome of the computation it represents. A fork node $f \in F$ does not represent computation; instead it merely passes control to all of its control successors. We call them fork nodes to emphasize that they represent concurrency; other authors call them "region nodes," although they mean the same thing.

In addition to being rooted and acyclic, the structure of the directed graph (V, C) satisfies two important constraints.

The predicate least common ancestor rule (PLCA) requires that for any node $n \in V$ with two different control paths to it from the root, the least common ancestor (LCA) of any pair of distinct predecessors of n is a predicate node. PLCA ensures that there is at most one active path to any node. If the LCA node was a fork, control could conceivably follow two paths to n, implying multiple executions of the same node, something we explicitly
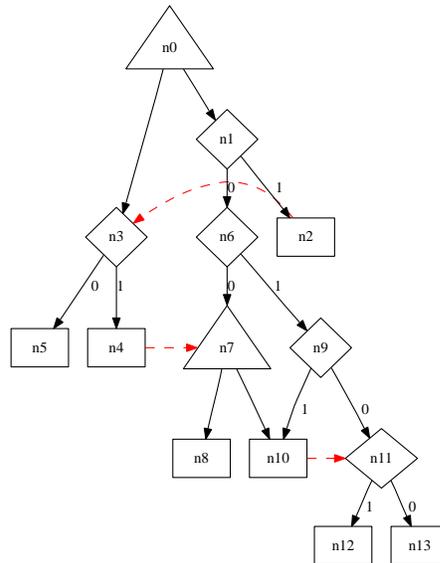
Figure 3.2: A program dependence graph requiring interleaving. Diamonds are predicate nodes, triangles are forks, and rectangles are statements. Solid lines are control arcs; dashed lines are data.

wish to prohibit.

The no-post-dominance rule: if *n* is a descendant of a node *m*, then there is some path from *m* to some statement node that does not include *n*. The rule holds because we insist that the PDG has eliminated unnecessary control dependencies among nodes. Otherwise, *m* and *n* would have been placed under a common fork.

3.2 Restructuring and Generating Code

3.2.1 Scheduling

Building a sequential control-flow graph from a program dependence graph requires ordering the concurrently running nodes in the PDG. In particular, the children of each fork node are semantically concurrent but must be executed in some sequential order. The main challenge is dealing with cases where data dependencies among children of a fork force

procedure PriorityDFS(*n*)

    if *n* has not been visited **then**

        add *n* to the visited set

        for each control successor *s* of *n* **do**

            PriorityDFS(*s*)

            $A[n] = A[n] \cup A[s]$

        for each control successor *s* of *n* **do**

            ComputeSuccPriority(*n*, *s*)

        if *n* has any incoming or outgoing data arcs **then**

            add *n* to $A[n]$

Figure 3.3: Successor Priority Assignment. For each node *n*, the array *A* holds the set of control descendants of *n* (including *n* itself) that have any incoming or outgoing data arcs.

their execution to be interleaved.

Figure 3.2 shows a PDG that illustrates the challenge. In this graph, data dependencies require n3 to be executed after n2 and n7 to be executed after n4. Thus, the two subgraphs under node n0 cannot be executed one after the other; they must be interleaved. The generated code must ensure nodes n2, n3, n4, and n7 execute in that order. This example is fairly straightforward, but such interleaving can become very complicated in large graphs with lots of data dependencies and reconverging control flow such as that at node n10.

Duplicating certain nodes in the PDG of Figure 3.2 could produce a semantically equivalent graph with no interleaving but it also could cause an exponential increase in graph size. Instead, we restructure the graph and add predicates that test guard variables. Unlike node duplication, this introduces extra runtime overhead, but it produces much more compact code.

Our approach inserts guard-variable assignments and tests based on cuts implied by a topological ordering of the nodes in a PDG. A cut represents a switch from an incompletely scheduled child of a fork to another child of the same fork. It divides the nodes under a

branch of a fork into two or more subgraphs.

To minimize the runtime overhead introduced by this technique, we try to add few guard variables by making as few cuts as possible. Ferrante, Mace, and Simons [36] showed that to find the minimum number of cuts is an NP-complete problem, so we attempt to solve it cheaply with heuristics.

We first compute a schedule for the PDG and then follow this schedule to find cuts where interleavings occur. We use a heuristic to choose a good schedule, i.e., one implying few cuts, that tries to choose a good order in which to visit each node's control successors. We identify the cuts while restructuring the graph.

To improve the quality of the generated cuts, we use the heuristic algorithm in Figure 3.3 to influence the scheduling algorithm. It computes an order for control successors of each node that the DFS-based scheduling procedure in Figure 3.5 uses to visit these successors.

We assign each control successor a priority vector of three integers (p_1, p_2, p_3) computed using the procedure described below, and later visit the successors in descending priority order while constructing the schedule. We totally order priority vectors: $(p_1, p_2, p_3) > (q_1, q_2, q_3)$ if $p_1 > q_1$, or $p_1 = q_1$ and $p_2 > q_2$, or if $p_1 = q_1$, $p_2 = q_2$, and $p_3 > q_3$. For each node n, the A array holds the set of control descendants of n (including n itself) that have any incoming or outgoing data arcs.

The first priority number of s_i, the ith subgraph under a node n, counts the number of incoming data dependencies (Figure 3.4). Specifically, it is the number of incoming data arcs from any other subgraphs also under node n to s_i minus the number of outgoing data arcs to other subgraphs under n.

The second priority number counts the number of elements that "pass through" the subgraph s_i. Specifically, it decreases by one for each incoming data arcs from a subgraph s_j to a node in s_i with a node m that is a descendant of s_i that has an outgoing data arc to another subgraph s_k ($j \neq i$ and $k \neq i$, but k may equal j).

The third priority counts incoming and outgoing data arcs connected to any nodes in

procedure ComputeSuccPriority(n, s)

  $(a, b, c) = (0, 0, 0)$ {initialize priorities}

  if s has neither incoming nor outgoing data arcs **then**

    a = minimum priority number

    return

  for each $j \in A[s]$ **do**

    $x = 0$, $y = 0$

    for each data predecessor p of j **do**

      if there is a path from $n \rightsquigarrow p$ **then**

        increase a by 1

        if there is not a path $s \rightsquigarrow p$ **then**

          increase b by 1

      increase c by 1

    for each data successor i of j **do**

      if there is a path $n \rightsquigarrow i$ **then**

        decrease a by 1

      decrease c by 1

    if $x \neq 0$ **then**

      for each $k \in A[j]$ **do**

        for each data successor m of k **do**

          if $n \rightsquigarrow m$ but not $s \rightsquigarrow m$ **then**

            increase y by 1

    decrease b by $x \cdot y$

  set the priority vector of s to (a, b, c)

Figure 3.4: Priority Computation

procedure ScheduleDFS(n)

    if n has not been visited **then**

        add n to the visited set

        for each ctrl. succ. i of n in descending priority **do**

            ScheduleDFS(i)

        for each data successor i of n **do**

            ScheduleDFS(i)

        insert n at the beginning of the schedule

Figure 3.5: The Scheduling Procedure

sibling subgraphs. It is the total number of incoming data arcs minus the number of outgoing data arcs.

Finally, a node without any data arc entering or leaving its descendants is assigned a minimum first priority number. Very likely, this kind of node does not need be involved in any interleaving. By assigning a minimum priority to it, we try to schedule the node to run as early as possible and therefore to minimize the cost of any possible interleaving.

The priority vector is meaningful only between a node and its control successors. For a node s that has multi-predecessors, its priority vector can be different considering each predecessor.

Under these definitions, the priority of the left successor under n0 in Figure 3.2 is $(0, -1, 0)$, and that the right successor is $(0, 0, 0)$. Arcs from n2 to n3 and from n4 to n7 both affect the first priority number, but their effects cancel out. The path n2 $\rightarrow$ n3 $\rightarrow$ n4 $\rightarrow$ n7 affects the second priority number of the left branch. Under our definitions, the right branch has highest priority and will be visited first during the depth-first search used for scheduling.

Similarly, node n9 will be visited before n7 because the first priority number of n7 is smaller due to the data arc n10 $\rightarrow$ n11. Finally, n5 will be visited after n4 because n5 has minimum priority.

The scheduling algorithm (Figure 3.5) uses a depth first search to topologically sort the

1: **procedure** Restructure

2:     Clear the currently active branch of each fork

3:     Clear master-copy(n) and latest-copy(n) for each node n

4:     **for each** n in scheduled order starting at the root **do**

5:         D = DuplicationSet(n)

6:         **for each** node d in D **do**

7:             DuplicateNode(d)

8:         **for each** node d in D **do**

9:             ConnectPredecessors(d)

Figure 3.6: The Restructure procedure.

nodes in the PDG. The control successors of each node are visited in order from highest to lowest priority (assigned by Figure 3.3). Ties are broken arbitrarily, and data successors are visited in an arbitrary order. The label on each node in Figure 3.2 indicates its position in the schedule: n1 is first, followed by n2, n3.

3.2.2 Restructuring the PDG

The scheduling algorithm presented in the previous section totally orders all the nodes in the PDG. Data dependencies often force the execution of subgraphs under fork nodes to be interleaved (control dependencies cannot directly induce interleaving because of the PLCA rule). The algorithm described in this section restructures the PDG by inserting guard variables (specifically, assignments to and tests of guard variables) according to the schedule to produce a PDG where the subgraphs under fork nodes are never interleaved.

The restructuring algorithm does two things: it identifies when a subgraph must be cut away from an existing subgraph according to the schedule and reattaches the cut subgraphs to nodes that test guard variables to ensure the behavior of the PDG is preserved.

The Restructure procedure (Figure 3.6) steps through the nodes in scheduled order, adding a minimal number of nodes to the graph under construction that ensures each node

in the schedule can be executed without interleaving the execution of subgraphs under any fork. It does this in three phases for each node. First, it calls DuplicationSet (Figure 3.7, called from line 5 in Figure 3.6) to establish which nodes must be duplicated in order to reconstruct the control flow to the node n. The boundary between the set D and the existing graph can be thought of as a cut. Second, it calls DuplicateNode (Figure 3.8, called from line 7 of Figure 3.6) on each of these nodes to create new predicate nodes that reconstruct control using a previously cached result of the predicate test. Finally, it calls ConnectPredecessors (Figure 3.9, called from line 9 of Figure 3.6) to connect the predecessors of each of the nodes in the duplication set, which incidentally includes n, the node being synthesized.

The main loop in Restructure (lines 4–9) maintains two invariants. First, each fork maintains its currently active branch, i.e., the successor in whose subgraph a node was most recently added. This information, tested in line 10 of Figure 3.7 and modified in line 7 of Figure 3.9, is used to determine whether a node can be added to an existing part of the new graph or whether the paths leading to it must be partially reconstructed to avoid introducing interleaving.

The second invariant is that the latest-copy array holds, for each node that appears earlier in the schedule, the most recent copy of each node. A node n can use these latest-copy nodes if they do not come from forks whose active branch does not lead to n.

The DuplicationSet function (Figure 3.7) determines the subgraph of nodes whose control flow must be reconstructed to execute the node n. It is a depth-first search that starts at the node n and works backward to the root. Since the PDG is rooted, all nodes in the PDG have a path to the root node and therefore DuplicationVisit traverses all nodes that are along any path from the root to n.

A node n becomes part of the duplication set D under three circumstances. The first case, tested in line 10, occurs when the immediate predecessor p of n is a fork but n is not the currently active branch of the fork. This indicates that to execute n would require interleaving because the PLCA rule tells us that there cannot be a path to n from p through

1: **function** DuplicationSet(n)

2:     $D = \{n\}$

3:     Clear the visited set

4:     DuplicationVisit(n)

5:     **return** D

6: **function** DuplicationVisit(n)

7:     **if** n has not been visited **then**

8:         Mark n as visited

9:         **for each** predecessor p of n **do**

10:             **if** p is a fork and $p \rightarrow n$ is not currently active **then**

11:                 Include n in D

12:             **if** latest-copy(p) is undefined **then**

13:                 Include n in D

14:             **if** DuplicationVisit(p) **then**

15:                 Include n in D

16:     **return** true if $n \in D$

Figure 3.7: The DuplicationSet function. A node is in the duplication set if it is along a path from a fork node that leads to n but whose active branch does not.

the currently active branch under p.

The second case, tested in line 12, occurs when the latest copy of a node is undefined. This occurs when a node is duplicated but its successor is not. The latest-copy array is cleared in lines 18–20 of Figure 3.8 when a node is copied but its successors are not.

The final case, line 14, occurs when any of n's predecessors are also in the duplication set.

As a result, every node in the duplication set D is along some path that leads from a fork node f to n that goes through a non-active branch of f, or leads from a node that has not been copied "recently." These are exactly the nodes that must be duplicated to reconstruct

all paths to n.

Once the DuplicationSet function has determined which nodes must be duplicated to reconstruct the control paths to node n, the DuplicateNode procedure (Figure 3.8) actually makes the copies. Duplicating statement or fork nodes is trivial (line 3): the node is copied directly and the latest-copy array is updated (line 21) to reflect the fact that this new copy is the most recent version of n, something that is later used in ConnectPredecessors. Note that statement nodes are only ever duplicated once, when they appear in the schedule. Fork nodes may be duplicated multiple times.

The main complexity in DuplicateNode comes when n is a predicate (lines 5–17). The first time a predicate is duplicated (i.e., the first time it appears in the schedule), the master-copy array entry for it is undefined (it was cleared at the beginning of Restructure—line 3 of Figure 3.6), the node is copied directly, and this copy is recorded in the master-copy array (lines 6–7).

After the first time a predicate is duplicated, its duplicate is actually a predicate node that tests v_n, a variable that stores the decision made at the predicate n (line 9). There is just one special case: the second time a predicate is copied (and only the second time—we do not want to add these assignments more than once), assignment nodes are added under the first copy (i.e., the master-copy of n in the new graph) that save the result of the predicate in the v_n variable. This is done in lines 11–13.

An invariant of the DuplicateNode procedure is that every time a predicate node is duplicated, the duplicate version of it has a new fork node placed under each of its successors (line 17). While these are often redundant and can be removed, they are useful as an anchor point for the nodes that cache the results of the predicate and in the uncommon (but not impossible) case that the successor of a predicate is part of the duplicate set but that the predicate is not.

Once DuplicateNode runs, all nodes needed to run n are in place but unconnected. The ConnectPredecessors procedure (Figure 3.9) connects these duplicated nodes to the appropriate nodes.

```
 1: procedure DuplicateNode(n)
 2:    if n is a fork or a statement then
 3:       Create a new copy n′ of n
 4:    else {n is a predicate}
 5:       if master-copy(n) is undefined then {making first copy}
 6:          Create a new copy n′ of n
 7:          master-copy(n) = n′
 8:       else {making second or later copy}
 9:          Create a new node n′ that tests vₙ
10:          if master-copy(n) = latest-copy(n) then {second copy}
11:             for i = 0 to (the number of successors of n) − 1 do
12:                Create a new statement node a′ assigning vₙ = i
13:                Attach a′ to the ith successor of master-copy(n)
14:          for each successor f′ of master-copy(n) do
15:             Find a′, the assignment to vₙ under f′
16:             Add a data-dependence arc from a′ to n′
17:          Attach a new fork node under each successor of n′
18:    for each successor s of n do
19:       if s is not in D then
20:          Set latest-copy(s) to undefined
21:    latest-copy(n) = n′
```

Figure 3.8: The DuplicateNode procedure. This makes either an exact copy of a node or tests cached control-flow information to create a node matching n.

1: **procedure** ConnectPredecessors(n)

2:    Let $n' =$ latest-copy(n)

3:    **for each** predecessor p of n **do**

4:        Let $p' =$ latest-copy(p)

5:        **if** p is a fork **then**

6:            Add a new successor $p' \rightarrow n'$

7:            Mark $p \rightarrow n$ as the active branch of p o

8:        **else** {p is a predicate}

9:            **for each** arc of the form $p \rightarrow n$ **do**

10:               Let f' be the corresponding fork under p'

11:               Add a successor $f' \rightarrow n'$

Figure 3.9: The ConnectPredecessors procedure. This connects every predecessor of n appropriately, possibly using nodes that were just duplicated. As a side effect, it remembers the active branch of each fork.

For each node n, ConnectPredecessors adds arcs from its predecessors, i.e., the most recent copies of each. The only minor trick occurs when the predecessor is a predicate (lines 9–11). First, DuplicateNode guarantees (line 17 of Figure 3.8) that every successor of a predicate is a fork node, so ConnectPredecessors actually connects the node to this fork, not the predicate itself. Second, it can occur that a single node can have a particular predicate node appear two or more times among its predecessors. The *foreach* loop in lines 9–11 connects all of these explicitly.

Running this procedure on Figure 3.2 produces the graph in Figure 3.10. The procedure copies nodes n1–n5. At this point, n0 $\rightarrow$ n3 is the active branch under n0, which is not on the path to n6, so a cut is necessary. DuplicationSet returns {n1, n6}, so n1 will be duplicated. This causes DuplicateNode to create the two assignments to v1 under n1 and the test of v1. ConnectPredecessors then connects the new test of v1 to n0 and n6 to the test of v1. Finally, the algorithm just copies nodes n7–n13 into the new graph.

Figure 3.11 illustrates the operation of the procedure on a more complicated example.
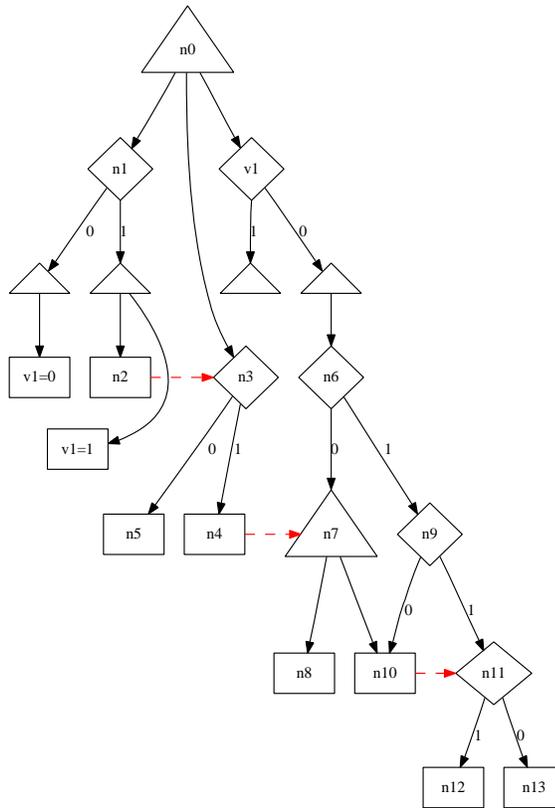
Figure 3.10: The restructured PDG from Figure 3.2. This example only adds the single guard variable v1. Some unary fork nodes generated by Restructure have been omitted for clarity.
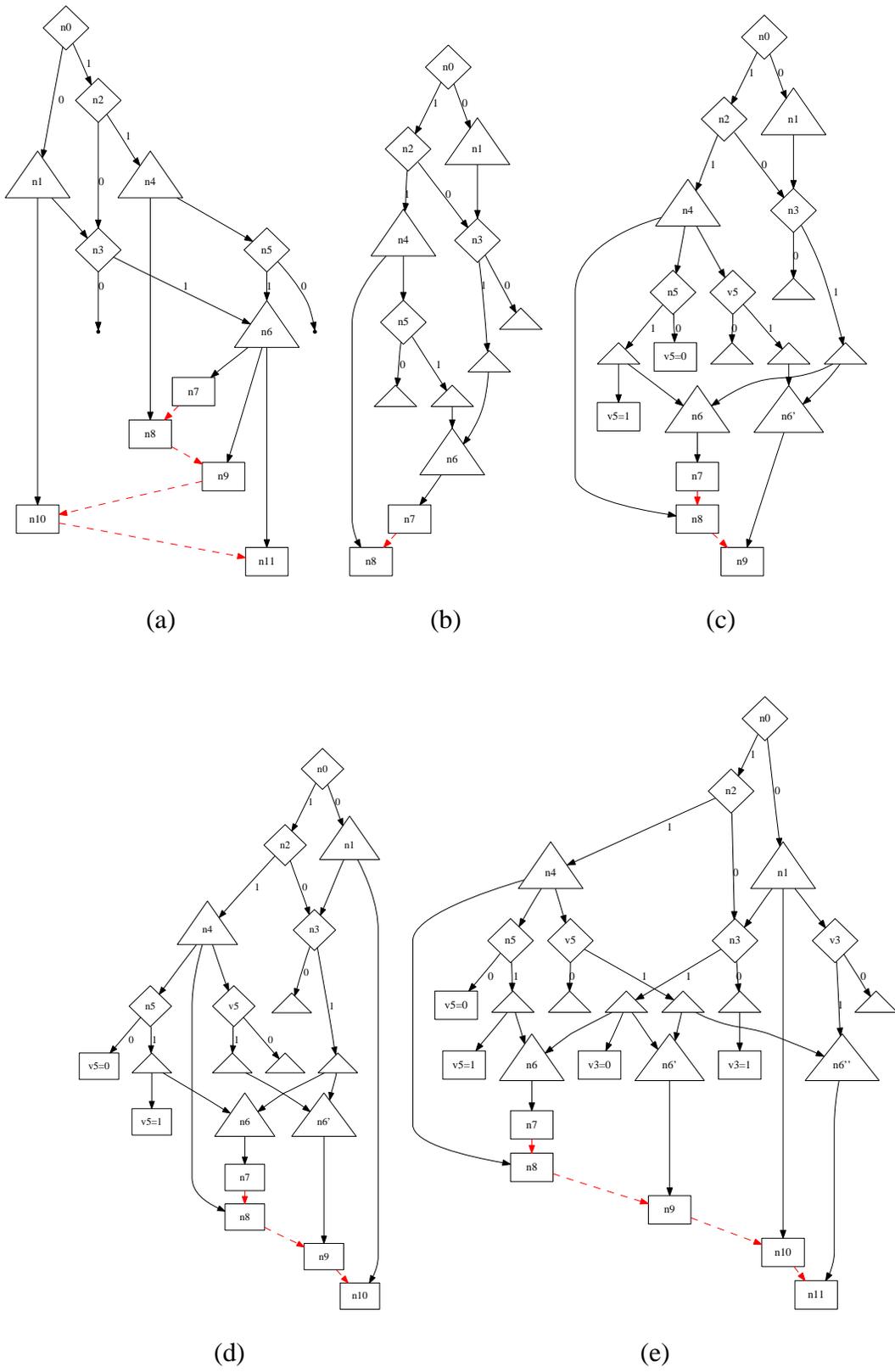
Figure 3.11: (a) A complex example. (b) After adding nodes n0–n8. (c) After adding n9, (d) n10, and (e) n11.

The PDG in (a) has some bizarre control dependencies that force the nodes to be executed in the order shown. The dizzying number of forced interleavings generates a fairly complex final result, shown in Figure 3.11e.

The algorithm behaves simply for nodes n0–n8. The state after n8 has been added is shown in (b).

Adding n9, however, is challenging. DuplicationSet returns {n9, n6, n5} because n8 is the active node under n4, so DuplicateNode copies n9, makes a second copy of n6 (labeled n6′), creates a new test of v5, and adds the assignments to v5 under n5 (the fork under the "0" branch from n5 has been omitted for clarity). Adding n9's predecessors is easy: it is just the new copy of n6, but adding n6's predecessors is more complicated. In the original graph, n6 is connected to n3 and n5, but only n5 was duplicated, so n6′ is connected to v5 and to a fork off the copy of n3.

Figure 3.11d adds n10, which is simple because although n3 was the active branch under n1, n10 only has it as a predecessor.

Finally, (e) shows the addition of n11, completing the graph. DuplicationSet returns {n11, n6, n3}, so n3 is duplicated and assignment nodes to v3 are added. Again, n6 is duplicated to become n6″, but this time n3 was duplicated.

An unfortunate choice of schedule clearly illustrates the need for guard variable fusion. Consider the correct but non-optimal schedule n0, n1, n2, n6, n9, n3, n4, n5, n7, n8, n10, n11, n12, n13 for the PDG in Figure 3.2. Figure 3.12 depicts the effect of so many cuts. The main waste is the cascade of conditionals along the right side of the graph (predicates on v1, v6, and v9). For efficiency, we replace such predicate cascades with single multi-way conditionals.

Figure 3.13 illustrates the effect of fusing guard variables. The predicate cascade has been replaced by a single multi-way branch that tests the fused guard variable v169 (formed by fusing predicates v1, v6, and v9). Similarly, group assignments to these variables are fused, resulting in three single assignments to v169 instead of three group concurrent assignments to v1, v6, and v9.
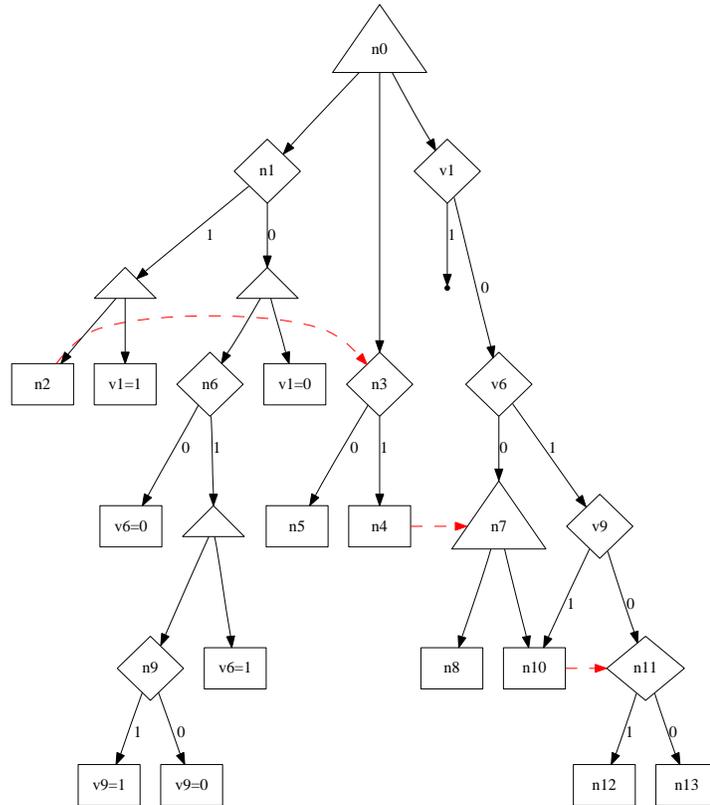
Figure 3.12: The reconstructed PDG from Figure 3.2 induced by a different schedule.

3.2.3 Generating Sequential Code

After the restructuring procedure described above, the PDG is in a state where the sub-graphs under each fork node can be executed in a particular order. This order is non-obvious when there is reconvergence in the graph, and appears to be costly to compute. Fortunately, Simons and Ferrante [67] developed the external edge condition (EEC) as an efficient way to compute this ordering. Basically, the nodes in eec(n) are executed whenever any node in the subgraph under n is executed.

In what follows, $X < Y$ denotes $G(X)$ must be scheduled before $G(Y)$; $X > Y$ denotes $G(X)$ must be scheduled after $G(Y)$; $Y \sim X$ denotes any order is acceptable; $Y \neq X$ denotes no order is acceptable. Here, $G(n)$ represents n and all its control descendants, i.e., all nodes in n's subgraph.
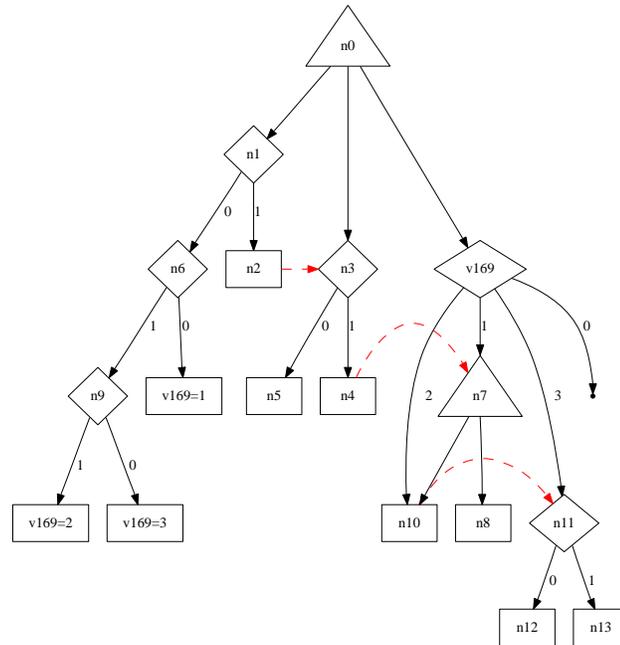
Figure 3.13: The PDG of Figure 3.12 after guard variable fusion.

We reconstruct the graph by ordering fork successors. Given the EEC information, we use the rules in Steensgaard's decision table [68] to order pairs of fork successors. When the table says any order is acceptable, we order the successors based on data dependencies. However, if, say, the EEC table says $G(X)$ must be scheduled before $G(Y)$, yet the data dependencies indicates the opposite order, the data dependencies win and two additional nodes are inserted, one that sets a guard variable and the other that tests it. Figure 3.14 illustrates the procedure.

In Figure 3.10, data dependency forces n11 > n10, but the external edge condition could require n10 > n11 if there were a control arc from a descendant of n11 to a descendant of n10 (i.e., if there were more nodes under n10). In this case, n10 ≠ n11, so our algorithm will cut the graph at n11 and add a guard there.

This produces a sequential control-flow graph for the concurrent program. We generate structured C code from it using the algorithm described in Edwards [30].

procedure OrderSuccessors(G)

  for each node n **do**

    if n is a fork node **then**

      original-successors = control successors of n

      clear the control successors of n

      for each X in original-successors **do**

        for each control successor Y of n **do**

          if $X \sim Y$ **then**

            if $\exists (m, n) \in D, m \in G(X), n \in G(Y)$ **then**

              insert X before Y in n's successors

          else if $Y < X$ **then**

            if $\exists (m, n) \in D, m \in G(Y), n \in G(X)$ **then**

              Cut Y

              insert X before Y in n's successors

          else if $Y > X$ **then**

            if $\exists (m, n) \in D, m \in G(X), n \in G(Y)$ **then**

              Cut X

            else

              insert X before Y in n's successors

          else if $Y \neq X$ **then**

            if $\exists (m, n) \in D, m \in G(X), n \in G(Y)$ **then**

              Cut Y and insert X before Y in n's successors

            else

              Cut X

        if X was not inserted **then**

          append X to the end of n's successors

Figure 3.14: The successor ordering procedure

Example	Lines	Average cycle times		
		Esterel V5	Lists	PDG
atds-100	948	45s	7.7s	1.3s
tcint	687	11s	2.8s	2.4s
multi6	113	10s	2.3s	1.4s
multi8	62	1.1s	1.7s	0.63s
greycounter	82	6.0s	3.9s	0.94s
abcd	111	5.2s	1.5s	1.7s

Table 3.1: Experimental Results

3.3   Experimental Results

We compared the speed of the code generated by our technique to that from the stock Esterel V5 compiler, which translates the Esterel program into a logic circuit and generates a program that simulates it; and the other C code generator in the Columbia Esterel Compiler (CEC), which produces statically scheduled discrete-event-like code dispatched by multiple linked lists [34].

To obtain the average cycle times shown in Table 3.1, we ran the generated C code from all three compilers (compiled with gcc -O3) for 10 million cycles on a 2.5 GHz Intel Pentium 4 running Linux. Most examples are fairly small, but tcint and atds-100 (both bus controllers) are reasonably large and, we believe, illustrative of our technique.

3.4   Related Work

Many techniques to compile Esterel have been proposed during the language's twenty-year history. Berry and Cosserat [10] were the first. They translated each program into a flat automaton by directly interpreting the operational semantics of the language. This technique was fairly time-consuming, but produced efficient code at the expense of size: the generated code may be exponentially larger than the source, making it impractical for

all but the smallest programs.

Next, Gonthier, as part of his thesis [41], devised a more efficient way to generate the automaton by simulating a control-flow-graph-like representation known as ıc. This formed the basis for the successful V3 compiler [11], but did not mitigate the exponential code size problem.

The V5 generation of Esterel compilers [8] translated Esterel programs into circuits, topologically sorted the gates, then generated simple code for each gate. While this technique scales much better than the automaton compilers, it does so at a great cost in speed. The fundamental problem is that the program must execute code for every statement in every cycle, even for statements that are not currently active.

Further progress in code generation came in 1999, when Edwards [29] and a group at France Telecom [12] independently developed two techniques that produced much faster code that was roughly the same size as that from the circuit-based compilers. The techniques we describe here are direct descendants of these two approaches.

Potop-Butucaru [63], as part of his 2002 PhD thesis [62], developed a much-improved version of the circuit-based code generation technique, incorporating a number of very clever optimizations to improve the quality of the generated code.

Ferrante and Mace [36] were the first to propose an algorithm for generating sequential code from an acyclic PDG, but their technique only works when no node duplication (or equivalently, the addition of predicates) is necessary.

Later, Simons and Ferrante [67] presented an efficient algorithm for generating sequential code from an acyclic PDG. Their major contribution is a technique for computing "external edge" information for each node and using this during the synthesis procedure. The input to their algorithm is limited to a graph with only control dependencies; they assume data dependencies have somehow been incorporated into the control dependencies.

Building on Simons and Ferrante's work, Steensgaard [68] removed the requirement that the control dependencies in the PDG be acyclic, thereby allowing loops in the generated code (earlier work assumed that loops had somehow been removed), but still assumed

that the generated code did not require either node duplication or the insertion of additional predicates. We have not integrated Steensgaard's cyclic extensions because they were unnecessary in our application.

Our technique extends Simons and Ferrante's in two ways. First, we propose a cutting algorithm that restructures the PDG and inserts additional predicate nodes before it is passed to Simons and Ferrante's basic algorithm, making it work for all valid acyclic PDGs. Second, we consider data dependencies to generate correct code for all valid PDGs.

Nacul and Givargis recently presented a code partitioning technique [59] for sequentializing multitasking C programs. The compiler first groups the basic blocks of task functions into disjoint clusters, then adds preempting and resuming scheme code for switching among these clusters. Although their general approach is similar to ours, the specific synchronous model our technique applies to provides us opportunities to aggressively reorder the code, which could not have happened on general C programs.

Our procedure resembles Edwards' technique for Esterel [30]. However, our use of a PDG representation instead of Edwards' concurrent control-flow graph makes it possible to rearrange independent statements among concurrent processes and further reduce context-switching overhead.

3.5   Summary

To demonstrate the potential of partial evaluation to optimize DSL programs, we have presented a PE algorithm that produces efficient sequential code from acyclic program dependence graphs generated from synchronous programs. Our technique, which consists of a heuristic scheduler followed by an exact restructuring procedure, produces sequential code while inserting a minimal number of guard assignments and tests, leading to faster execution with fairly low overhead when compared to existing mechanisms.

Experimentally, we have shown that this algorithm produces efficient code when applied to the synchronous, concurrent language Esterel.

Although this partial evaluation technique was applied specifically to Esterel, it should be applicable to other synchronous, concurrent languages.

Chapter 4

Partial Evaluation for Separate Compilation

Synchronous models are useful for designing real-time embedded systems because they provide timing control and deterministic concurrency. A synchronous system is composed of modules that communicate with each other and march in step to a global clock. However, problems arise when there are communication dependency cycles among modules. It is difficult to compile and simulate in isolation a module involved in cycles, since only partial input information is available for it. Therefore, the semantics of synchronous paradigm require an entire system to be compiled at once to make it possible to analyze the dependencies among modules. The alternative is to write modules that can respond when the values of some of their inputs are unknown, a tedious and error-prone process to do manually.

This chapter provides a concrete example of applying a specialized PE technique to solve a complex issue in DSL compilation. We present a partial evaluation technique that enables modules in a synchronous system to be compiled separately, even when the system has communication cycles. This automatic process allows a programmer to describe synchronous modules without having to consider undefined inputs. Our algorithm transforms such a description into code that does as much as it can with undefined inputs, allowing modules to be compiled separately and assembled later. This work originally appeared in

the proceedings of ICESS'2005 [78].

The chapter is organized as follows. We start by introducing the problem and the challenges. Then, we define the Graph Representation Code (GRC), an IR generated from the synchronous program to be analyzed. All examples in this chapter are represented in GRC. After that, we demonstrate the algorithm with the help of an example. The experimental results are analyzed at the end and we propose some future work. Also, we compare our solution with other related work.

4.1 Compilation and Assembly of Concurrent Systems

The synchronous model of computation [7] has emerged as a successful, practical way to assemble models of concurrent embedded systems because of its deterministic concurrency and its precise control over time. Each process in a synchronous model operates in lockstep with a global clock, and communication between modules is implicitly synchronized to this clock. Provided the processes execute fast enough, processes can precisely control the time (i.e., the clock cycle) when something happens.

In addition to domains including avionics [9] and hardware design [3], the synchronous model has been used for constructing processor simulations [73, 61]. Especially in this latter setting, heterogeneous synchronous models [35], which can assemble and run synchronous components with no knowledge about their contents, is preferable because it allows separate compilation of components (e.g., cache models, branch prediction units) and even allows them to be written in different programming languages.

In the heterogeneous synchronous model [35], a system is assembled from a collection of concurrently running blocks that communicate through instantaneous "wires" each connected from a single block's output port to one or more input ports on other blocks. That the blocks be able to respond when not all their input wires are defined is the main requirement for being able to run such blocks without knowledge of their contents. Furthermore, a block must be well-behaved when presented with unknown inputs, e.g., if a block decides

output o has value v even though input i is undefined, it may not change its mind, e.g., change the output to w once i becomes defined. But if blocks do obey these rules, such a system can adopt a Ptolemy-like philosophy [17] in which systems can be assembled from black-box components and executed efficiency with precise, deterministic semantics.

Although it is possible to write such well-behaved synchronous blocks in a general-purpose language such as C, it is a tedious and error-prone process. The alternative, which we propose here, is for the programmer to write blocks only taking into account their behavior when all their inputs are applied and have the compiler interpolate the correct behavior of the block when only some of the inputs are applied. While it would be correct to make the blocks strict, i.e., to respond with no information about any output unless all the inputs are defined, this is not very helpful.

In this chapter, we propose an algorithm that does this interpolation on programs written in the synchronous concurrent, imperative language Esterel [11]. Constructs in Esterel only explicitly address the behavior when all inputs are known (i.e., the user cannot control them to respond in a certain way to unknown values), but their semantics are clear when not all inputs are known.

Our work generates code from Esterel that responds to unknown inputs. The enables separate compilation and the assembly of modules written in other languages.

4.2 The Graph Code Representation

We represent the programs we are compiling using a variant of the Graph Code (GRC) format due to Potop-Butucaru [63]. GRC is like a traditional control-flow graph augmented with concurrency and nodes for controlling it. However, loops are prohibited (cross-cycle loops are allowed). The result is a compact, precise way to represent Esterel programs [11], which we compile with our technique, although the same representation could be used for other synchronous, imperative languages.

A GRC program $G = (N, r, c, V, O, S, t)$ is similar to the program dependence graph

(PDG) defined in Chapter 3 but with more details. It includes a set of nodes N and a distinguished root r ($r \in N$). Control successors function c is the same to that in PDG.

The finite set V denotes variables. $O \subset V$ are the output variables. $V \setminus O$ are the input variables. S denotes the set of possible states of the program.

Each node has a type given by the function $t : N \rightarrow$ {assign-v-to-one, assign-v-to-zero, predicate-on-v, fork, switch, enter, terminate-at-l, sync }. When executed, an assign-v-to-one node sets the variable v to 1 (v is a variable in V). Predicate-on-v tests variable v and sends control to one of its successors; switch is similar but tests program state instead of a variable; enter changes the program state. A fork node sends control to all its successors, which must eventually re-converge at a sync node. All predecessors of a sync must be terminate-at-l nodes, which indicate the exit level of their respective threads. A sync node passes control to the successor whose number corresponds to the highest-numbered terminate node that passed control to it.

The assign-v-to-zero nodes are only added to the graph during our construction. As its name suggests, an assign-v-to-zero node sets the variable v to 0. In two-valued execution, a variable's default value is 0, making such nodes unnecessary. But in the three-valued execution that is the result of our procedure, variables default to the undefined value and therefore require assign-v-to-zero nodes.

Figure 4.1 depicts such a program graphically. All arcs point downward. The type of each node is indicated by its shape. Assignments are boxes, predicates are diamonds, forks are triangles, terminates are octagons, and syncs are upside-down triangles. The label on a predicate or assignment node indicates the variable tested or set. For predicate nodes, the first (false-valued) arc is indicated with a bubble at its source. The label on a terminate indicates the exit level of the corresponding thread. For sync node, each arc is labeled with a number that matches the exit level. A dashed line denotes a data dependency (as shown in Figure 4.1b: $6 \rightarrow 10$, $9 \rightarrow 10$, $10 \rightarrow 14$, $15 \rightarrow 16$).

A two-valued execution of a GRC program (which contains no assign-to-zero nodes by definition) starts with an initial program state and an assignment of values to input variables

(i.e., either $v = 0$ or $v = 1$ for all $v \in V \setminus O$). Then it derives a subset S of the nodes as follows. S includes the root node; every successor node of each fork, assignment, enter, or terminate node in S; and for every predicate node n in S that refers to variable v, the first (true) successor is in S if v is an input variable with value 1 or the graph includes an assignment-to-one node for v, and the second (false) successor of n otherwise. For a sync node, all of its predecessors' (terminate nodes) exit levels are checked, and S includes the sync's successor under the branch whose label is the same as the highest exit number. The value of each output variable is 1 if the set includes an assignment-v-to-one node to variable v and 0 otherwise.

Consider executing the graph in Figure 4.1a using the node numbers from Figure 4.1b and with the assignments A=1, B=1, C=0, and D=1. Node 1 is in S since it is the root, and since A=1, node 2 is also. This adds nodes 3 and 8. Since B=1, node 12 is in S but node 9 and node 11 are not, and since C=0, node 4 is in S, and since D=1, node 6 and 7 are in S but node 5 is not. Since node 7 and node 12 are included, and node 7's exit level (1) is higher than node 12 (0), sync node 13's branch 1 is executed. That excludes node 14 and 15 from S. In the end, $S = \{1, 2, 3, 4, 6, 7, 8, 12, 13\}$ so E=1 and F=0.

The above procedure requires the value of every input variable to be known when the program starts; we want to relax this. In particular, if we know the values of only certain inputs, we would like to conclude whatever we can about as many outputs as possible provided they are consistent with any future values for the unassigned inputs.

One way to answer this question is to execute the GRC program using three-valued logic, i.e., adding a third value that represents unknown or undefined (we write it $\perp$) to the usual 0s and 1s. This introduces another set of nodes to the simulation procedure: those that might run if additional input is provided later. The three-valued simulation is a more complicated procedure that does not reduce to the usual sequential execution behavior of imperative programs, unlike the two-valued simulation of GRC defined above, which can be transformed into sequential code using a fairly inexpensive procedure as described in Chapter 3.

4.3 Generating Monotonic Three-Valued Programs

Our main contribution is the algorithm described here that takes a GRC program and constructs a fast sequential program that evaluates the graph in the three-valued domain, i.e., it allows some of the input variables to be undefined. Our algorithm works in four phases (see Figure 4.3). Given a GRC program, we add nodes and arcs to represent data dependencies, compute a topological order of this annotated graph, compute information about the subgraph under each node that will tell us what information we can forget during a simulation of the program, and finally construct a sequential program by performing this simulation. We try to keep the size of the generated program under control; we do this by allowing as much reconvergence as possible in the generated code, i.e., by identifying (and reusing) equivalent states during the simulation.

4.3.1 Adding Data Dependencies

The algorithm starts by adding data dependencies. For each output variable v, this process adds an assign-v-to-zero node and then adds arcs from each assign-v-to-one node to this new node, and arcs from this new node to each predicate-on-v node that tests v. The result is that there is now a path from each assign-v-to-one node for a variable to each node that tests that variable, hence ensuring the topological sort respects data dependencies. Furthermore, it introduces an assign-v-to-zero node that will appear in the schedule when it is possible to determine that a particular variable may be zero. Figure 4.1b shows the effect of applying this procedure on Figure 4.1a.

4.3.2 Summarizing Dependency Information

Keeping the size of the generated graph under control is the main trick in our algorithm. Although it would be correct to consider the value of each variable and control arc when considering which subgraphs can be shared during code generation, this would be very inefficient and always produce an exponentially large tree as a result. Instead, we attempt

to model the state of a simulation using as little information as possible because we want to consider a maximum number of states to be identical so code for them can be shared.

Our insight is this: at a particular point in the schedule, we only care about nodes that appear later in the schedule since by definition we must have already executed anything earlier, and only two things matter about these nodes: the variables they test and the state of control arcs that lead from nodes earlier in the schedule to later nodes.

Consider building a subgraph for the nodes starting at 8 in Figure 4.1b, and assume the node numbers correspond to their position in the schedule. At this point, the simulation will have established values for variables A, C, and D, but we do not directly care about any of them since code for them has already been generated and we will not test any of them later. However, we do care about whether node 10 will be executed, which can be affected by node 6, and whether node 13 was triggered by its predecessors, since we will be generating code for nodes 10 and 13 (they appear after 8 in the schedule).

As a result, we consider identical any simulation states that differ only on variables A, C, or D. We also consider the control flowing in to nodes 8, 10, and 13.

The ComputeRelavantVars procedure (Figure 4.4) builds two sets that exactly capture this notion of which variables and control states we care about during the construction. By stepping through the nodes of the graph in scheduled order, ComputeRelavantVars computes relevant_arcs[s_i], the set of all arcs that go from nodes before s_i in the schedule s to nodes after s_i, and relevant_vars[s_i], the set of all variables that are either tested or set in the nodes after s_i. Note that because s is a topological order, nodes after s_i in the schedule necessarily include the subgraph under s_i.

In Figure 4.1b, if $s = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$, ComputeRelavantVars finds relevant_arcs[8] = {2→8, 6→10, 5→13, 7→13}, relevant_vars[6] = {B, E, F}. Both relevant_vars and relevant_arcs are global and are not modified after ComputeRelavantVars.

4.3.3 Construct

The Construct procedure (Figure 4.5) simulates the three-valued behavior of the GRC program and, as a side-effect, constructs our objective: a graph that reproduces its behavior. In addition to the node n that is being synthesized, it takes three arrays: val[v] is the value (0, 1, or $\perp$) of each variable; ctrl[n, i], $i = 0, 1, \ldots$ is the state (again, 0, 1, or $\perp$) of each control arc leaving each node; and term[n, i] is the state of each termination level $i = 0 \ldots M$ reaching each sync node n (M is the maximum possible exit level reaching n).

Construct begins by checking for an end condition: for the last node in the schedule s, the "node following it" is simply null. It then computes two partial functions (associative arrays): var_state, which contains the value of each relevant variable, i.e., those set or tested by any node that comes after n in the schedule (computed earlier by ComputeRelavant-Vars); and node_state, which computes the execution state (1=will run, 0=will not run, or $\perp$=might run) of all the relevant nodes, i.e., predecessors of n plus all those with incoming arcs that come before n in the schedule (again, computed earlier by ComputeRelavantVars).

Together, the node itself and the two partial state functions constitute the total state on which the subgraph to be built for n. The procedure then looks to see whether a subgraph with identical state has already been built and returns it if it exists.

Otherwise, the real work starts. First, the node following n in the schedule is identified as m, since it will be recursed on later. The procedure assumes the node n is a flow-through type (e.g., assign-v-to-one or a fork) and sets all its control successors to have the same activation condition as the node itself. These assignments will be modified below when necessary, especially for predicate and switch nodes.

There are two main cases: once the node is known not to run, this information is propagates as far as possible by the PropagateZeros procedure. Nodes that set each such variable to zero are created, assembled into a chain. Finally the subgraph that executes the nodes after n is connected to the end of this chain after a recursive call to Construct.

The other case, when the node might or is known to run (node_state = $\perp$ or 1), is handled quite differently (Figure 4.6). Dealing with assign-v-to-one and enter nodes is simple: if it

is known to run, it is simply copied to the new graph. Furthermore, for an assign-*v*-to-one node, the value of *v* is set to 1 so that it will be propagated to later constructions.

Conditional nodes (predicate-on-*v* and switch) are more complicated. To deal with them, the BuildCondition function is called (Figure 4.7). If the processed node *n* is a predicate-on-*v* and *v*'s value is known, the branches under *n* are set to active and inactive depending on the value.

Otherwise, if the node is a switch or a predicate-on-*v* whose variable *v* is unknown, the algorithm constructs an identical conditional node in the generated program and considers all possibilities: one of the branches—corresponding to a possible condition—is set active, and the others are made inactive (their control state is set to zero). For switch, the possible conditions correspond to each of its successors. For a predicate node, the possible conditions are related to the variable's value, which can be true, false, or unknown when the generated program runs. In the last condition, all branches are set active. For each condition, the variable value is saved appropriately in val array and then Construct is called on the next node in sequence with the new state.

Terminate and sync nodes deal with exit levels and are handled separately. For every sync node, its related threads' exit levels are preserved by the term array. When a terminate-at-*l* node is met at the end of a thread, if it is known to be executed, it sets the term array element of the exit level *l* to be 1 for the corresponding sync; if its control value is $\perp$ and no other thread exited at the same level, the element in the term array is set to $\perp$. The sync node computes the highest possible exit level(s) by looking at the term array, then passes its control value to the corresponding branch. This algorithm simulates the two-valued behavior. BuildSync in Figure 4.8a simulates sync's behavior.

For all these types, Construct is called on the next node *m* and saves the root of returned subgraph to *n″*. Switches and predicates are exceptional: they have different new states built to meet all possible conditions, so Construct is called for every condition.

Finally, *n′* is the new node as the root of the subgraph constructed on *n*. To make it possible to later identify its state, this fact is recorded in BuildNode. *n′* is returned to the

caller, which probably adds an arc leading to it.

We use a few simple helper functions (not shown). Link(n,m) connects arcs: if n is null, it returns m; otherwise, a control arc $n \rightarrow m$ is added and n is returned. Copy(n) creates a new node in the generated program with the same type and variable as node n.

4.3.4 State

The Construct procedure maintains a collection of subgraphs in the generated program, each corresponding to a particular node in the original program and the state that it implicitly assumes the original program was in before reaching the subgraph. Such a state is a triple: $\langle n, \text{var\_state}, \text{node\_state} \rangle$. n is the node leading the subgraph constructed, var_state is a partial assignment of values to variables the subgraph cares about, and node_state is an analogous assignment of values to control arcs relevant to the subgraph. Specifically, those that pass into the subgraph from outside: arcs within the subgraph, by definition, will be evaluated as part of the subgraph.

4.3.5 Monotonicity

The code generated by our algorithm is monotonic. When adding data dependencies (Section 4.3.1), an assign-v-to-zero node is linked after all assign-v-to-one and before all predicate-on-v nodes. This ensures assign-to-one nodes appear first in the topological order, followed by the assign-to-zero node, and finally all predicates that test v.

A $v = 0$ assignment is made only when none of the assign-v-to-one nodes could or did execute (see Figure 4.5 line 17-18 and Figure 4.8b), so the code will never change a variable's value from 1 to 0. It is also impossible for the generated code to change v's value from 0 to 1 because the topological ordering of nodes places assign-to-ones before assign-to-zeros. The *val* array records variables' values throughout the Construct function. So when a predicate-on-v node is met (see Figure 4.6 line 16-17 and Figure 4.7), val[v] is checked first. If v's value is known, the only active branch will be set, and the val[v] will

not be touched but just passed to later construction (see Figure 4.7 line 2-5).

4.3.6 The Example

Figure 4.9 illustrates some of the algorithm's behavior on Figure 4.1b. A, B, C, and D are input variables; E and F are outputs. Figure 4.1b was derived from Figure 4.1a by adding data dependencies. Figure 4.9a shows the graph after assuming $A=\perp$, $C=0$, $D=0$, and $B=0$ and arriving at node 14. The label on each arc indicates its value in the ctrl array. Figure 4.9b is similar, but it assumes $A=\perp$, $C=1$ and $B=\perp$ (predicate-on-D is known not to run in this configuration, so D's value is irrelevant). Our algorithm determines that the code generated for these two states is the same and can be shared.

Specifically, at node 14, variables E and F are relevant (and unknown in both Figures 4.9a and 4.9b) and the state of node 14 is relevant. In both cases, the state of 14 is $\perp$, which is equal to the ctrl value of incoming arc 13→14.

In these two states, node 10 may still run in the future, so no code is generated to set E to 0, E is therefore also unknown, so it is tested, and F=0 may later be able to run. The code generated for these states is the test of E followed by the assignment of F to 0 in the dashed region of Figure 4.2. Paths from the test of C (i.e., when C is 0—Figure 4.9a) and the test of B (i.e., when B is $\perp$—Figure 4.9b) converge on this subgraph because the algorithm has identified these states as equivalent.

By contrast, assuming $A=\perp$, $C=0$, $D=0$ and $B=1$ gives the state in Figure 4.9c. Here it is known that node 10 (assign 0 to E) will run because none of its predecessors will (this is reversed from the usual rule because such nodes are specially designed to detect when a variable is set to 0). This leads to different code of the other two cases, i.e., the assignment of 0 to E attached to the true branch under the test of B in Figure 4.2.

Example	Lines	Average cycle times		
		Esterel V5	**SCFG**	**3-Valued**
comexp	88	1.67s	0.61s	0.80s
iwls3	70	1.04s	0.35s	0.26s
3vsim2	48	0.68s	0.32s	0.46s
multi3	120	1.39s	0.45s	0.47s

Table 4.1: Experimental Results

4.4 Experimental Results

We compared the speed of the code generated by our algorithm to that from the Esterel V5 compiler, which translates the Esterel program into a logic circuit and generates code to simulate it, and to the code generated by the algorithm described in Chapter 3, which generates sequential code by adding guard variables. To obtain the average cycle times in Table 4.1, we ran the generated C code from all three compilers (compiled with gcc -O3) for 10 million cycles on a 2.5 GHz Pentium 4 running Linux.

Table 4.1 shows our results. While the theoretical complexity of our algorithm is exponential, the experiments we ran show it appears to not be an issue in practice for modest-sized program.

The code generated by the other two compilers (V5 and SCFG) only perform two-valued computation. Because our compiler adds code for three-valued computation, it generates slower code. However, the experimental results suggest that the slow-down is fairly mild and in some cases, our compiler actually generates faster code. We suspect it is because our compiler uses a different technique to sequentialize the concurrent code.

Together, these experiments suggest that our algorithm is practical for modest-sized programs. There are certainly additional opportunities for optimization. In particular, we intend to integrate this technique with our earlier technique for producing efficient sequential code from (concurrent) program dependence graphs (Chapter 3).

4.5   Related Work

Digital logic simulators often perform a similar two- to three-valued interpolation. In hardware description languages such as Verilog or VHDL, users often compose systems out of apparently two-valued logic functions such as AND or OR. The simulator, however, interprets them as three-valued functions and performs the simulation in the extended domain. It has long been known, however, that this tends to greatly slow the simulation and attempts have been made to circumvent it where possible (e.g., by detecting when two-valued-only simulation is possible and doing it when possible). Overcoming this speed penalty is a primary goal of our work.

Our intermediate representation bears some resemblance to binary decision diagrams (BDDs—see, e.g., Bryant's survey [16]), but differ enough to make their manipulation very different. Compared to the most common type of BDD, the ROBDD (reduced, ordered BDD), our programs may test variables in different orders and multiple times along a path. Although certain styles of BDDs (e.g., free BDDs) relax this restriction, our formalism is even less like most BDDs because it can communicate within itself, i.e., assign and later test the value of the variable assigned, whereas BDDs typically only make assignment at their leaves. As a result, most BDD algorithms, which are able to assume disciplined variable orderings and a single type of node, are inapplicable for our application. Others, however, have used BDDs to synthesize software [19].

Our algorithm is like a partial evaluation of a three-valued simulator on programs represented as graphs, which resembles many other techniques for generating sequential code from concurrent models [31]. Our algorithm, as a side-effect, orders the nodes under forks and generates a purely sequential program. While this is probably undesirable for certain systems, more clever techniques, such as the one described in Chapter 3 could probably be woven into this three-valued simulator to more efficiently generate sequential code.

4.6 Summary

To illustrate applying partial evaluation to solve complex issues in compiling DSLs, we presented a PE technique for compiling synchronous modules separately. Although our algorithm was originally designed to generate monotonic three-valued programs from two-valued ones to work with the heterogeneous synchronous model of computation, it can have other applications. The general idea of partially simulating networks and recording the results as a branching program resembles some approaches for generating efficient simulators for gate-level circuit descriptions [58, 5]. While these approaches use a BDD-like representation, our technique suggests the possibility of selectively "forgetting" inputs, which gives an interesting trade-off between efficiency and code size.

The experimental results show our algorithm is practical for modest-sized programs. However, it does not work well for large programs and may generate exponential code. We expect to solve this problem by further research.
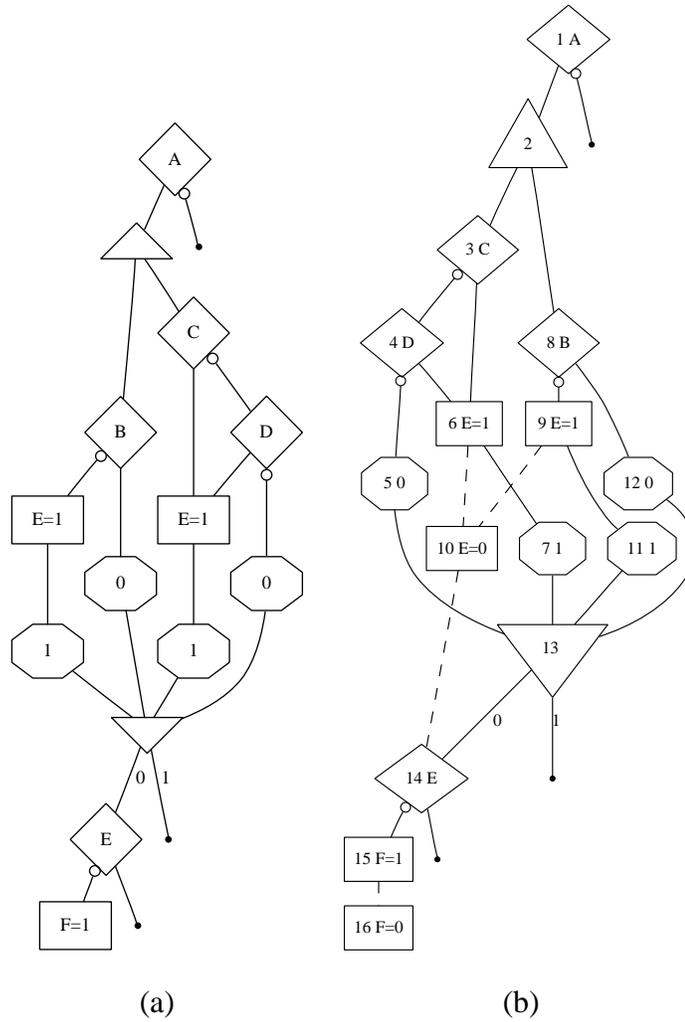
Figure 4.1: (a) A two-valued GRC. Arcs with bubbles are taken when a variable is 0. (b) After adding data dependence nodes and arcs to it.
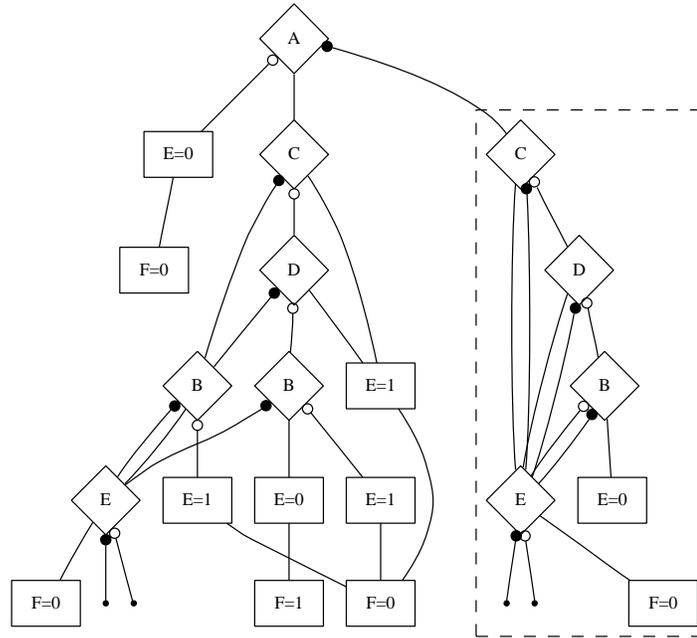
Figure 4.2: Three-valued projection of the GRC in Figure 4.1(a), produced by our algorithm. Arcs with solid bubbles are taken when a variable's value is unknown. Figure 4.9 shows the construction of the nodes in the dotted region.

procedure Main(G)

　Add data dependencies

　s = topological sort of the augmented graph

　ComputeRelavantVars()

　Set val[v] = ⊥ for all variables

　Set ctrl[n, i] = ⊥ for all nodes & successors

　Set term[n, i] = ⊥ for all sync & exit lvls

　Construct(root of G, val, ctrl, term)

Figure 4.3: The Main procedure

procedure ComputeRelavantVars()

    for $i = 1, \ldots, N$ **do** {schedule is $s_1, \ldots, s_N$}

        Set relevant_arcs$[s_i] = \emptyset$

        Set relevant_vars$[s_i] = \emptyset$

        for each $j = i, \ldots, N$ **do**

            for each arc $s_k \rightarrow s_j$ with $k < i$ **do**

                add $s_k \rightarrow s_j$ to relevant_arcs$[s_i]$

            if s_j tests or set any variable v **then**

                add v to relevant_vars$[s_i]$

Figure 4.4: ComputeRelevantVars

1: **function** Construct(n, val, ctrl, term)

2:     **if** n is null **then**

3:         **return** null {bottom of the program}

4:     Clear node_state {partial function on nodes}

5:     node_state[n] = 1 if n is the root node, $\perp$ otherwise

6:     **for each** arc $p \xrightarrow{i} q$ in relevant_arcs[n] **do**

7:         node_state[q] = node_state[q] OR ctrl[p, i]

8:     var_state = val {partial function on variables}

9:     **for each** v not in relevant_vars[n] **do**

10:         var_state[v] = DONTCARE

11:     **if** BuiltNode[$\langle n$, var_state, node_state$\rangle$] exists **then**

12:         **return** BuiltNode[$\langle n$, var_state, node_state$\rangle$]

13:     m = node following n in s {on which to recurse}

14:     **for each** successor n_i of n **do** {assume flow-through}

15:         ctrl[n,n_i] = node_state[n]

16:     **if** node_state[n] = 0 **then** {node known not to run}

17:         PropagateZeros(n, node_state, ctrl, val)

18:         Create chain $(v_1 = 0) \rightarrow (v_2 = 0) \rightarrow \cdots \rightarrow (v_k = 0)$ for each v_i where val[v_i] = 0

19:         Add an arc from $(v_k = 0) \rightarrow$ Construct(m, val, ctrl,term)

20:         n' = the first node in the chain: "$(v_1 = 0)$"

21:     **else** {node_state[n] is $\neq 0$}

22:         n' = MakeNode(n, val, ctrl, term node_state)

23:         **if** $n.type$ is not switch or predicate-on-v **then**

24:             n'' = Construct(m, val, ctrl,term)

25:             Link(n', n'')

26:     BuiltNode[$\langle n$,var_state,node_state$\rangle$] = n'

27:     **return** n'

Figure 4.5: The Construct Function

1: **function** MakeNode(n, val, ctrl, term, node_state)

2:    $n' = $ NULL

3:    **case** $n.type$ **of**

4:    Assign-v-to-one **:**

5:      **if** node_state[n] = 1 and val[v] = $\perp$ **then**

6:        $n' = $ Copy(n) {assign-v-to-one that executes}

7:        val[v] = 1

8:    Enter **:**

9:      **if** node_state[n] = 1 **then**

10:        $n' = $ Copy(n) {Enter that executes}

11:    Terminate-at-l **:**

12:      $c = $ SyncMap[n] {the sync node related with n}

13:      term[c][l] = term[c][l] OR node_state[n]

14:    Sync **:**

15:      BuildSync(n,ctrl,term)

16:    Switch or predicate-on-v **:**

17:      $n' = $ BuildCondition(n, m,val,ctrl,term)

18:    **return** n'

Figure 4.6: The MakeNode Function

1: **function** BuildCondition(n, m,val,ctrl,term)

2:    **if** n is predicate-on-v and val[v] is known **then**

3:       ctrl[n,val[v]] = node_state[n] {active branch}

4:       ctrl[n, $1-$val[v]] = 0 {inactive branch}

5:       n' = Construct(m, val, ctrl,term)

6:    **else** {switch or predicate with unknown variable}

7:       n' = Copy(n)

8:       **for each** successor n_i of n **do**

9:          ctrl[n, i] = node_state[n] {active branch}

10:          **for each** successor n_j of n other than n_i **do**

11:             ctrl[n, j] = 0 {inactive branch}

12:             **if** v is not NULL **then** {predicate value is $\perp$}

13:                val[v] = i

14:          Add an arc $n' \rightarrow$ Construct(m, val, ctrl,term)

15:       **if** n is a predicate **then**

16:          **for each** successor n_i of n **do**

17:             val[v] = $\perp$

18:             ctrl[n, i] = node_state[n] {active branches}

19:          Add an arc $n' \rightarrow$ Construct(m, val, ctrl,term)

20:    **return** n'

Figure 4.7: The BuildCondition Function.

function BuildSync(n,ctrl,term)

  unknown_ctrl = false

  findmax = false

  for each i in term[n], max to

  min **do**

    if term[n][i] is 0 **then**

      ctrl[n][i] = 0;

    else

      if findmax is false **then**

        findmax = true

        if term[n][i] is $\perp$ **then**

          unknown_ctrl = true

      if unknown_ctrl is true

      then

        ctrl[n][i] = $\perp$

      else

        ctrl[n][i] =

        node_state[n]

    if term[n][i] is 1 **then**

      break

  return ctrl

function PropagateZeros(n,

              node_state, ctrl, val)

  if n is null **then**

    return

  node_state$'$ = node_state

  for each arc $t \xrightarrow{i} n$ **do**

    node_state$'$[n] =

      node_state$'$[n] OR ctrl[t, i]

  if node_state$'$[n] is 0 **then**

    for each successor n_i of n **do**

      ctrl[n, i] = 0

    if $n.type$ is Assign-v-to-zero **then**

      val[v] = 0

  m = node following n in s

  PropagateZeros(m, node_state$'$, ctrl, val)

      (a)   (b)

Figure 4.8: (a) The BuildSync Function and (b) the PropagateZeros function
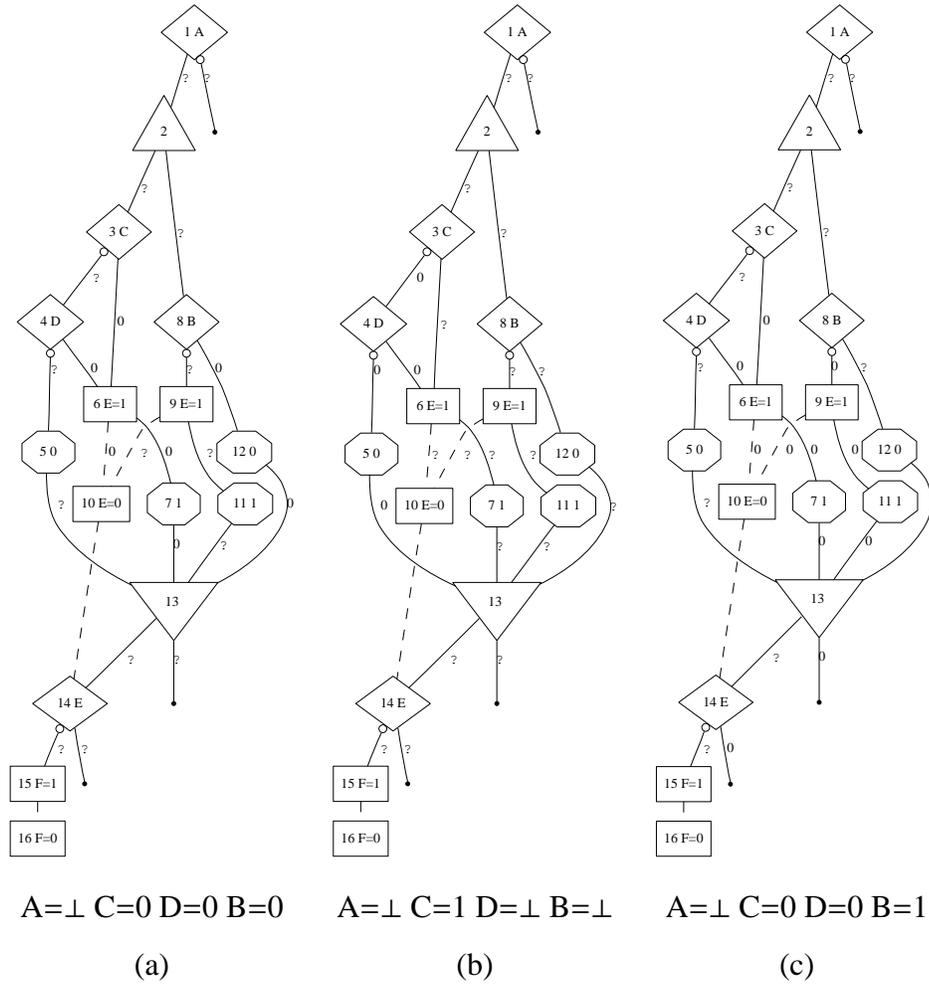
Figure 4.9: (a), (b), (c) Possible simulation states upon reaching node 14. Cases (a) and (b) are equivalent since the relevant variables (E and F) and state of node 14 (the one with incoming arc(s) from outside of the subgraph) are the same. Case (c) is different. Cases (a) and (b) share the node that tests E, whereas case (c) creates the E=0 node in the dashed box in Figure 4.2.

Chapter 5

Partial Evaluation for Unrolling Recursion

Hardware design requires static specifications, yet dynamic software facilities like recursion provide flexibility and may be useful in hardware design. So there is a need to for removing recursion from specifications. Static elaboration enables a compiler to transform the dynamic description of a system to a static implementation. A concurrent language for system design might use recursion as a convenient way to create concurrent structures or processes. Static elaboration tries to statically evaluate such dynamic structures and thus allows users to algorithmically build up a concurrent system. The compiler then transforms the resulting static description of the system by instantiating necessary components. However, not all dynamic structures can be evaluated statically. Therefore we use partial evaluation to elaborate. It enables the same language to define both static and dynamic operations and have the recursive structures compiled away.

In this chapter, we introduce a partial evaluation technique that is able to unroll a recursive program, allowing it to be implemented in hardware. We begin with our motivation. Then we present a static elaboration technique that analyzes at compile time the call graph of a program with mutually recursive functions to produce a "flattened" result that uses only bounded resources, often inlining functions for efficiency. To illustrate the algorithm, we

demonstrate an elegant implementation of a pipelined FIFO in SHIM with recursive function calls. Also, we include an implementation of FFT using recursive calls in Appendix B, which is succinct yet predictable in terms of resources. Finally, we show the experimental results that compare the numbers of functions/threads before and after unrolling.

5.1 Compilation of Recursive Programs

Widely adopted in software languages, recursion provides an elegant solution to complex and even infinite computations and data structures. It is good for divide-and-conquer algorithms, which can also be solved using traditional iterative structures. The advantage of the recursive approach is the succinct way it defines the problem and it is usually easier to verify. One disadvantage, however, is the run-time space complexity. The Towers of Hanoi problem, for example, can be solved in three lines with recursion (two recursive function calls and one move), but requires a call stack of 3^n, where n is the number of disks to move.

Because of its potential infinite behavior, recursion is rarely used in embedded software, which usually has much stronger predictability requirements. Often, the number of function calls in a design must be static, which is not obvious or guaranteed in a design with recursion.

SHIM [69], a concurrent language targeted at embedded software and hardware, provides recursive function calls. This interesting language aspect enables SHIM users to create parallel structures through recursion. Comparing to the stiff way of listing these structures in a parallel statement, parametrized recursive call is a more succinct and flexible solution.

To enable the language to define both static and dynamic operations, the SHIM compiler statically elaborates recursive calls and thus bounds the space complexity and maximum number of function calls in a design.

We present the static elaboration algorithm that we use in the SHIM compiler. The algorithm works in two steps. First, it decomposes the control-flow of the program into a

collection of (mostly) tail-recursive functions, which are the extended basic blocks in the program. Then, following the call graph of these functions, it performs constant propagation and unrolls loops of mutually recursive functions by making variant copies of them. Using this technique, the elaborator effectively estimates the space complexity of the program and unfolds it.

5.2   Static Elaboration

The static elaboration algorithm removes recursion from a program when the recursion depth can be bounded. Ideally, the elaborator will generate a new program whose function call graph (CG) is acyclic. This procedure is nontrivial since recursion implies cycles in the call graph. If the elaborator fails, i.e., the CG remains cyclic after the transformation, we fall back on the original recursive program and explain that we were not able to statically analyze the recursion. This may be a problem for bounding resources, but it means that we are still able to run the program even if the elaborator fails.

Figure 5.1 shows the main algorithm. Starting from the IR, the algorithm decomposes each function into extended basic blocks and then treats each such block as a (usually tail-recursive) function. Thus, the control flow graph is transformed into a function call graph whose root is the entry block of the main function. This step may greatly increase the recursion in the program since iteration is treated in the same way as recursion. Converting iterations into loops is for simplifying the algorithm presentation. Operating on the decomposed program, the algorithm propagates constants, values that can be statically determined, while unrolling the call graph. After unrolling, the algorithm cleans up the code by eliminating dead variables and removing unused function arguments. Also, functions with a single entry are inlined to minimize the final number of functions. This process transforms the IR in Figure 5.3(b) to Figure 5.5, which is exactly a five-stage parallel pipeline.

The unroll procedure, shown in Figure 5.2, is the key procedure in the algorithm. Starting with the root function, this procedure performs both inter-process and intra-process

constant propagation to generate a new set of functions without recursion. By intra-process propagation we mean to pass value between functions and blocks, which are treated as functions during analysis, through function calls and control flow between blocks respectively.

Each process, therefore, is decomposed into single-entry blocks with tree-structured control flow and similarly the program is decomposed into processes. The intra-process analysis becomes straightforward since no loops need to be considered. The inter-process analysis, on the other hand, iterates on a list of functions combined with actual parameters, called *unrolling*. A recursive process may be unrolled several times. The algorithm generalizes the parameters to either a constant value or not a constant (NAC). For each pair of a function and an assignment of actual parameters to it, a unique version of the function is made by propagating the actual values throughout the function. The new function is added to the *unrolled* set, mapping from the pair of original function and actual parameters. All these new functions compose the reconstructed program.

The unroll iteration continues until no new actual parameter assignment is found for any function, or when the number of actual parameter sets found for a function exceeds a user defined limit. To avoid generating exponentially large code, the algorithm creates a new function only when the original function is called with a new set of actual parameter values. A function called at different places in the original program with the same set of values is not duplicated. After unrolling, a design whose call graph contains mutually recursive functions is either completely expanded to a directed acyclic graph or is left cyclic because the recursion is truly unbounded (e.g., the depth is dependent on run-time data) or because it requires more function duplication than we are willing to allow.

Constant propagation in the unrolling procedure is specialized for SHIM. Since a interface variable represents a channel, even when it is known to be constant during the analysis, it cannot be simply replaced with the constant number in a function call statement. Otherwise the called function will lose track of the channel it communicates through. However, the channel may become redundant if all values it passes in are constants and it is not con-

nected to any subchannels. In this case, it can be eliminated from the interface and replaced with concrete numbers in related expression. The channel n in Figure 5.3(a), for example, can be removed. Like a loop index, it is served for control flow, i.e., being used to determine when the recursive construction terminates. When the recursive structure is statically elaborated, this channel is useless and will be eliminated.

The procedure for eliminating unused channels aims at those variables, which in SHIM are channels. The rule is simple. A channel c of a function f will be removed from f's formals if a constant is passed in through c and c is not passed as parameter to any function called in f, and also no run time communication (e.g. recv c) is found on f. Such a c will be transformed into a local variable and assigned a constant value passed in. The parameter n in fifo_0() in Figure 5.4, for example, will be become a local variable with a constant value of 3. Thus, redundant channels are removed from the function declarations after this step.

5.3   Unrolling a Pipelined FIFO in SHIM

SHIM provides a cohesive model for both hardware and software designs, with deterministic concurrency. It follows a C-like syntax where a program is composed of functions. Neither global variables nor pointers are allowed. Instead, SHIM includes a mechanism for concurrent function calls through the *par* construct and rendezvous-style inter-process communications through the *next* operator. Figure 5.3(a) shows a SHIM program that defines a five-stage FIFO. Recursive function calls, such as fifo() in the example, are allowed and can be combined with the *par* statement. These features enable SHIM's model to be expressive while being strict.

In the SHIM compiler, we use an intermediate representation (IR) based on three address code, inheriting the function-statement-expression hierarchy. Figure 5.3(b) shows the IR for the program in Figure 5.3(a). The formal parameters of a function declaration are identified as either inputs (plain) or outputs (labeled with &). Loops and predicate

procedure Main(program, limit)

    Decompose every function in program to extended basic blocks

    Make every block a function

    root = main entry function

    n = number of root's formal parameters

    vs = a set of length n whose elements are NACs

    unrolled = []

    unrolling = [(root, vs)]

    for each function f **do**

       unrolledtimes[f] = 0

    Unroll(limit, unrolled, unrolling, unrolledtimes)

    functions = functions defined in unrolled set

    Eliminate unused channels for all functions

    Eliminate dead code for each function

    Inline functions

Figure 5.1: The Main procedure.

procedure Unroll(limit, unrolled, unrolling, unrolledtimes)

   while unrolling set is not empty **do**

     (f, vs) = first pair in unrolling

     if (f, vs) is not found in unrolled **then**

       n = unrolledtimes[f]

       if (n + 1) > limit **then**

         f' = f

         goto END

       else

         n = n + 1

       vmap = empty

       ps = formal parameters of f

       for each p in ps **do**

         v = value corresponding to p in vs

         add p→v to vmap {v can be a const or NAC}

       f' = a renaming of f corresponding to (f, vs) {const propagation on f body}

       set f' body to empty

       for each statement s in f's body in order **do**

         (s', vmap') = ConstPropagation(s, vmap)

         add s' to f' body

         if s involves one or parallel function calls **then**

           for each g called in s **do**

             gvs = actual parameters g is called with

             add (g, gvs) to unrolling set if none exists

             g' = a renaming of g corresponding to (g, gvs)

             replace g with g' in s statement

       END: remove (f, vs) from unrolling

       add (f, vs)→f' to unrolled

Figure 5.2: The Unroll procedure.

```
void main() {
  int a;
  int b;
  int n = 3;

  source(a); par fifo(a, b, n); par sink(b);
}

void fifo(int i, int &o, int n) {
  int c;
  int m = n - 1;

  if (m)
    g(i, c); par fifo(c, o, m);
  else g(i, o);
}

void g(int b, int &c) {...}

void source(int &a) {...}

void sink(int b) {...}
```

```
main()
local int32 a
local int32 b
local int32 n
  n = 3
  source(a) : fifo(a, b, n) : sink(b);

fifo(int32 i, int32 &o, int32 n)
local int32 c
local int32 m
  m = n - 1
  ifnot m goto _else3
  g(i, c) : fifo(c, o, m);
  goto _endif4
  _else3:
  g(i, o);
  _endif4:

g(int32 b, int32 &c)

source(int32 &a)

sink(int32 b)
```

(a) SHIM Code   (b) IR Code

Figure 5.3: A FIFO program.

expressions are dismantled into statements, labels, and gotos.

The program in Figure 5.3 is constructing a pipelined FIFO where source() reads every input, sends it to fifo() through channel *a*; fifo() processes the data and put the result on channel *b*; sink(), which is waiting on channel *b*, finally collects the result and returns. In fact, fifo() is composed by some small concurrent processors g() that are pipelined. fifo() recursively constructs these instances of g() while constraining the pipeline size by *n*, which is a compile time constant in the example. For simplicity, we only show the details of the main and fifo functions in Figure 5.3, which we will use to illustrate our algorithm. We assume the others (g(), source(), and sink()) are simple functions without recursion.

In this section, we demonstrate our algorithm's operation on the recursive fifo() function

```
main()
local int32 a
local int32 b
local int32 n
  n = 3
  source__0(a) : fifo__0(a, b, n) : sink__0(b);

fifo__0(int32 i, int32 &o, int32 n)
local int32 c
local int32 m
  m = 2
  g(i, c) : fifo__1(c, o, m);

fifo__1(int32 i, int32 &o, int32 n)
local int32 c
local int32 m
  m = 1
  g(i, c) : fifo__2(c, o, m);

fifo__2(int32 i, int32 &o, int32 n)
local int32 c
local int32 m
  m = 0
  g(i, o)

source__0(int32 &a)

g(int32 b, int32 &c)

sink__0(int32 b)
```

Figure 5.4: The FIFO after unrolling

```
main()
local int32 a
local int32 b
local int32 c__fifo__0
local int32 c__fifo__1
  source__0(a) : g(a, c__fifo__0)
  : g(c__fifo__0, c__fifo__1)
  : g(c__fifo__1, b) : sink__0(b);

source__0(int32 &a)

g(int32 b, int32 &c)

sink__0(int32 b)
```

Figure 5.5: The FIFO after inlining

in the FIFO example (Figure 5.3).

Figure 5.6 shows the step by step procedure of decomposing and unrolling the program. Comparison of the two call graphs in Figure 5.6(b) and (f), before and after the static elaboration respectively, illustrates the effect of the unroll procedure which eliminates the original cycle of fifo::__L0 → fifo::__L1 → fifo::__L0.

Each extended basic block (EBB) (Figure 5.6(a)) is treated like a function, whose parameters are variables alive at the entry of the block. fifo::__L0, for example, has four formal parameters (Figure 5.6(b)). At the beginning, the main() function calls fifo() with $n = 3$, which is passed to fifo::__L0 while all its other formal parameters are set to NAC (Figure 5.6(c)). We use (*) to represent NAC in Figure 5.6, which refers to either unknown or not-a-constant value in our algorithm.

The unroll procedure maintains two sets during the iteration. The unrolling set keeps all pairs of function and parameters to be processed, whereas the unrolled set records all pairs which have been processed and the process result, i.e., the corresponding functions newly constructed after unrolling. By this, the elaborator can avoid duplicate work. It does not have to unroll a function/parameter pair that have been unrolled before. The functions

in the unrolled set and in unrolling are illustrated as solid and dashed nodes respectively in (Figure 5.6(c) - (e)). For fifo() example, the unrolling set starts with [(fifo::__L1, (*, *, *, 3))]. This only pair, being unrolled, generates a new function fifo_0::__L1(), which is added to *unrolled* as a mapping of (fifo::__L1, [*, *, 3])→fifo_0::__L1() (Figure 5.6(c)). This constant is propagated to the following functions fifo::__L1() and fifo::__L2(), which are added to the unrolling set. After evaluation with this set of value, a specified copy of fifo::__L1 is connected to fifo_0::__L0 (Figure 5.6(d)). The *ifnot* prediction is removed in fifo_0::__L1 since the elaborator knows that the par call branch will be taken in this case. Therefore, two other new pairs are added to the unrolling set considering that the pairs are not present in unrolled. The functions, i.e., EEBs, in fifo() iteratively add themselves to the unrolling set until $n = 1$ when in fifo::__L1 the else branch is taken. The resulting functions are shown in Figure 5.6(e) while Figure 5.6(f) illustrates the corresponding call graph, which obviously has no cycles.

All the unrolled functions are combined together and inlined as shown in Figure 5.5, compose into a program without recursive calls.

The unrolling process can also be used for hardware synthesizing with some extra work, such as to make all parallel function calls unique. A parallel call "g() : g()" is not a problem in software, but in hardware it is because every function will be synthesized to a processor which cannot run in parallel with itself. Therefore, the three parallel called instances of g() in main() in Figure 5.5 will be modified to be g__1():g__2():g__3() where these renamed functions are simply duplications of g().

5.4 Experimental Results

To experiment with our static elaborator, we implemented three algorithms in SHIM using recursion: the Fast Fourier Transform (FFT), square-root using Newton's Method, and the pipelined FIFO example from Section 5.3; the complete implementation of FFT in SHIM is presented in the appendix. We compared the number of functions in the original pro-
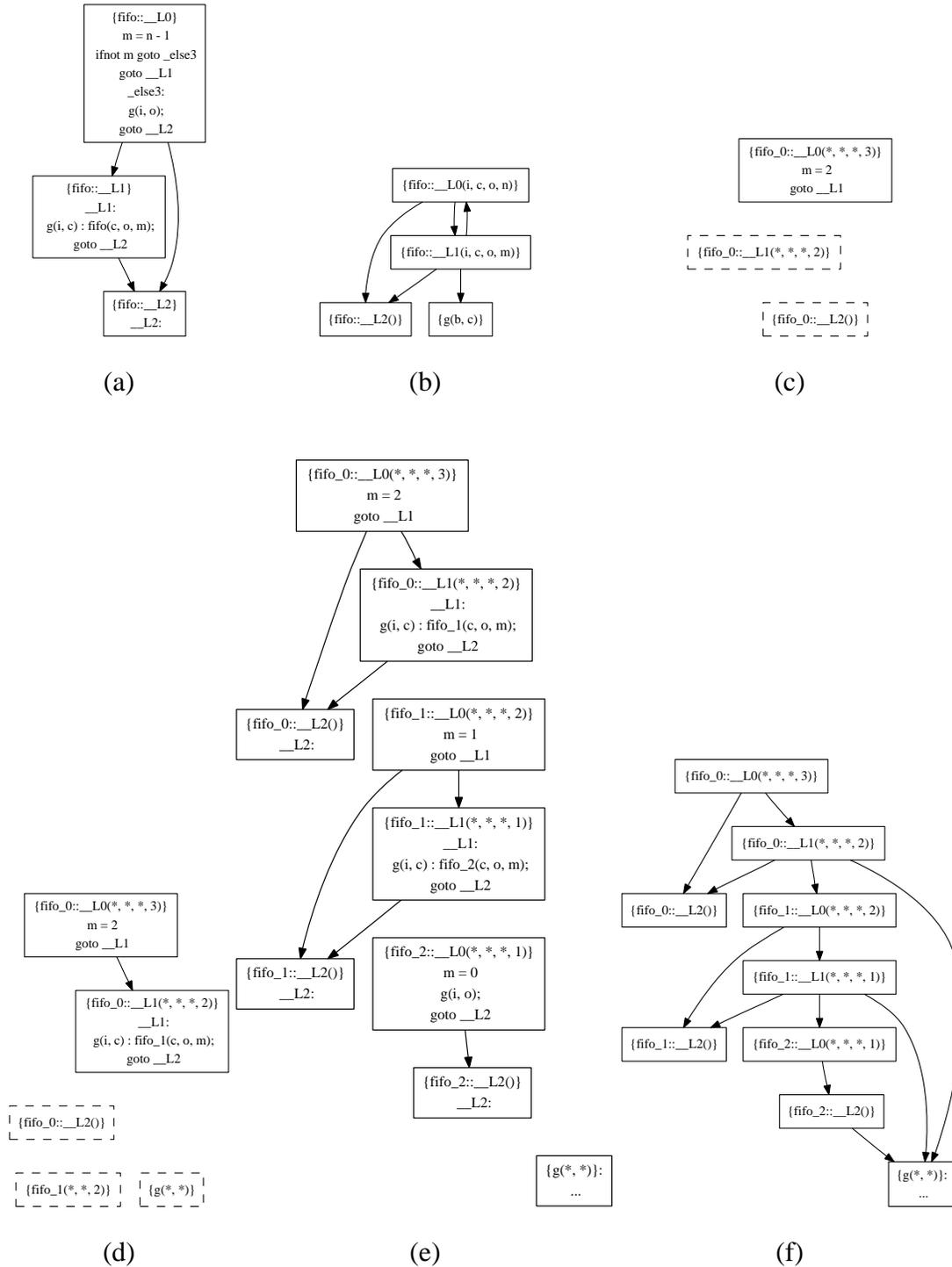
Figure 5.6: Unrolling the fifo(). (a) CFG of EBB after decomposition. (b) CG of decomposed functions. (c) After static elaboration on entry function fifo::__L0 with $n = 3$. Dashed nodes are functions in unrolling set whereas the solid ones are in unrolled set.(d) After elaboration on fifo::__L1 with $m = 2$. (e) CFG completely unrolled. (f) CG after unrolling.

gram, the unrolled program, and finally the inlined program, combined with the factor—a compile-time constant that defines the recursion depth. The factor for FFT, for an instance, is the input size. The results are shown in Table 5.1.

Depending on the factor, a program may grow fairly large after unrolling, as the "Unrolled" column shows. The reason is obvious: it exhaustively lists all functions customized with possible parameter sets. Most of the customized functions, however, simply pass control to their successor functions and thus are eliminated after inlining. Extreme cases, such as Sqrt and FIFO, can be simplified to a constant number of functions. On the other hand, the FFT only shrinks by half. Although most of the control functions remaining are very small in code size, they are necessary for the synchronization of parallel function calls and therefore cannot be eliminated.

Besides the factor, a user may set a limit that bounds the maximum number of times a function will be unrolled (Figure 5.2). To test its effect, we set the limit to 100 when unrolling FFT-128 and FFT-256. The result shows the numbers of functions generated become very close. For the rest of the experiment where the limit is not specified, we assume the number is big enough for the algorithm to fully unroll all functions. This limit setting helps user to effectively bound the maximum number of function calls in a design.

5.5   Related Work

Recursion is not a common feature of hardware languages, especially for synthesis. SystemC, a C++ like language for modeling hardware designs, allows recursion. Many synthesis tools for SystemC [66][42] provide static elaboration phases that cope with finite loops, but not with recursion, which is much harder to bound, especially when it involves mutually recursive calls. Another tool created by Moy etc. [56] uses static elaboration to extract high-level architecture information from SystemC programs. It establishes static structures in the program and creates their instances. However, it does not deal with dynamic structures such as pointers. Our approach is more general and takes care of both

Example	Number of Functions			Factor	Limit
	Org	**Unrolled**	**Inlined**		
FFT	9	37	20	8	
		919	455	128	
		2073	1032	256	
		132	61	128	100
		135	63	256	100
Sqrt	3	12	2	10	
		102	2	100	
		1002	2	1000	
FIFO	5	14	4	10	
		104	4	100	
		2004	4	2000	

Table 5.1: Experimental Results

static and dynamic operations.

Other few hardware languages that allow recursion include Hoe and Arvind's Bluespec [44], Li and Leeser's HML [55], and Bjesse et al.'s Lava [14]. All of these are based on functional languages. Bluespec provides very powerful static elaboration that deals with recursion. However, the language is quite different from SHIM. Its syntax decides that recursion is only used to describe sequential operations, while in SHIM it can be applied to concurrent structures.

Lava provides a framework based on Haskell for user to design and analyze their implementations on FPGAs. The sorting network designs shown in their work [21, 20] demonstrate how to describe recursive circuit in Lava. Because the language targets at FPGAs, it strictly requires that recursion be bounded at compile time. Our approach, by contrast, targets hardware and software designs. Therefore we deal with both bounded and unbounded recursive calls.

Rugina and Rinard [64] describe a recursion unrolling algorithm tailored to divide and conquer programs. Given a recursive function, the main idea is to recursively unroll this function to a certain depth by inlining previous unrolled version of the same function but with less depth. They use condition fusion to optimize the generated code. Our approach is more general because it handles mutually recursive function calls whose recursion depths are not given to the compiler. Also, because function calls may be parallel in SHIM, not every recursive function call can be inlined. The inlining process has to be done carefully.

5.6   Summary

To illustrate the method of customizing partial evaluation for a specific DSL, we presented a static elaborate algorithm for SHIM, a concurrent language providing recursive function calls that can be used to construct concurrent structures elegantly. For a SHIM program with recursion, the algorithm analyzes the program's call graph and unrolls the recursive calls to produce flattened code that uses bounded resources. This partial evaluation technique enables SHIM users to make their designs both succinct and resource-bounded.

Chapter 6

Conclusions

In this final chapter, we summarize our major contributions and outline some possible future work.

6.1   Contributions

Specialized partial evaluation can be effectively applied to solve compilation issues arising in domain-specific languages. The concise syntax of DSLs simplifies the development process and enables the compiler to have a more comprehensive understanding of the program's behavior. To take advantage of this and design a promising PE technique for a DSL, it is necessary to deeply understand the computational paradigm of the language. The challenge, therefore, is to create a customized PE technique that makes in-depth analysis of the language's semantics and fits well the specific paradigm. The results we achieved taking this approach are impressive.

We demonstrated three concrete examples of designing PE techniques to generate code from DSLs. We did not provide a general partial evaluator for all these languages because we observe that their radically different models require that each PE technique be specifically designed for that model to achieve interesting levels of optimization.

These three PE techniques addressed different essential issues in generating code from

concurrent, deterministic languages. We expect that their diversity will illustrate the potential of PE to solve different compilation issues for DSLs.

For simulating concurrency on a single-threaded processor, speed is one of the key factors. The algorithm for slicing concurrent programs presented in Chapter 3 provides a low-cost solution for it. By transforming a concurrent flow graph to a program dependence graph, the compiler exposes more concurrency in the program, which gives the scheduler more flexibility. It aggressively forms as many large atomic blocks as it can to minimize the time spent switching among them. Guard variables are added to store and recover states in the statically scheduled code. This technique has been successfully implemented in the Columbia Esterel Compiler. The experimental results show great speedup over existing techniques. However, how to apply the same methodology to other synchronous concurrent languages is not clear. Finding a way to dismantle a synchronous concurrent program to an acyclic flow graph with data dependence may be the first step.

We also considered the problem of separate compilation. Software engineering, which defines a systematic and disciplined approach to development, requires most designs to be split into modules. However, this can be difficult for synchronous programs since normally they can only respond to complete inputs. For a module that is involved in a communication cycle, some of its inputs may be unknown at run time. To make a synchronous module respond correctly to partially known inputs, we invented an algorithm to provide a preliminary solution to this issue, which we presented in Chapter 4. Through partial evaluation, the compiler explores all possible execution paths, allowing it to interpolate the correct behavior with unknown inputs and add this to the generated code. It works like a partial evaluation of a three-valued simulator. Again, we applied this algorithm to Esterel. The results show it is practical for modest-sized programs, but it does not work for large programs because the branching-like simulation may lead to an explosion in the number of reachable state when the decisions of early segment of code strongly affects later execution paths. We explain the situation more and propose a possible research direction in Section 6.2.

The third algorithm we presented (Chapter 5) statically elaborates recursive function

calls in a concurrent program. By compiling their programs using this static elaboration technique, programmers can use recursion to create concurrent components and have them instantiated at compile time. This enables the user to build up a concurrent system conveniently and algorithmically. The partial evaluation process involves constant propagation and procedure inlining, fairly typical compiler techniques. However, they are customized to fit SHIM: the asynchronous and concurrent DSL on which we experimented. The recursive FFT example we implemented in SHIM demonstrated that this PE technique is practical.

6.2 Future Work

Most of our algorithms could be improved further to make them more general and effective. As an example, we suggest how we might apply Program Dependence Graphs to separate compilation to better handle large programs.

The techniques we presented in this dissertation perform offline partial evaluation. However it may also make sense for DSLs to consider online partial evaluation, which benefits from knowing input information.

6.2.1 Separate Compilation of Large Synchronous Programs

The algorithm we proposed in Chapter 4 cannot deal with large synchronous programs because in it may generate exponentially large code that contains all possible sequential execution paths of the original concurrent program. The PDG technique we presented in Chapter 3 may help to avoid code explosion. We show two cases where using a PDG would generate smaller code than our current approach.

Figure 6.1 shows two cases that may cause an explosion in the size of the generated code. They are represented in graph code, which we defined in Chapter 4. The graph in Figure 6.1(a) is composed of a chain of predicates. Assume the decision of predicate *A* does matter in generating code that runs after predicate *B*. Following the rule described in Chapter 4, the compiler should not "forget" *A*'s decision before running *B*. Therefore,
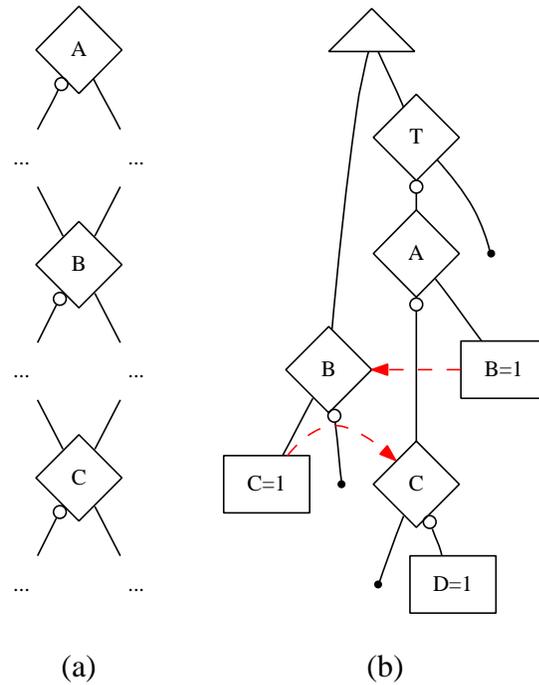
Figure 6.1: Graph examples that may generate exponential code using the algorithm in Chapter 4 (a) A chain of predicates. (b) Interleaving threads.

when constructing the new graph that explicitly defines actions for unknown inputs, our algorithm will make three copies of *B* under each control successor of *A* (will run, will not run, might run). Every copy of subgraph *B* will contain a subgraph for *C* if the same assumption applies to it. In such a sequence, a chain like Figure 6.1(a) will be expanded into a tree with exponentially many more nodes than the chain.

Figure 6.1(b), a fragment of graph code from the example program in Figure 2.1, is actually a variant of (a). Data dependence forces the two concurrent threads in the program to be interleaved. Therefore they actually run sequentially, much like the predicate chain in Figure 6.1(a) would. It follows that the graph constructed by the algorithm of Chapter 4 for this fragment may also grow exponentially large.

However, such an exponential increase in code can be avoided if the construction algorithm is applied after the program is translated into a PDG and context-switching code is added. Consider Figure 6.2, which shows the predicate chain in Figure 6.1(a) disman-
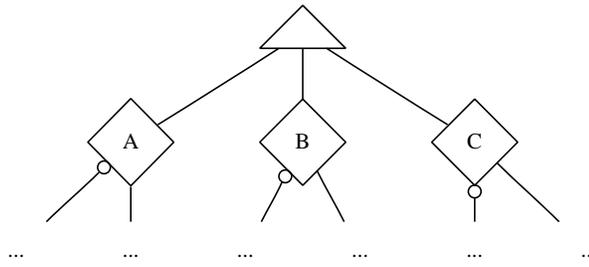
Figure 6.2: The PDG transformed from Figure 6.1(a)

tled into parallel predicate trees, each of which will expand to a small subgraph with three branches after applying the tri-branch construction algorithm. Once this is done, we can use the algorithm in Chapter 3 to sequentialize the constructed PDG. The size of the graph generated with this approach would be linear of the original graph size.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[2] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Second International Symposium on Static Analysis (SAS)*, pages 33–50, London, UK, 1995. Springer-Verlag.

[3] Laurent Arditi, Amar Bouali, Hedi Boufaied, Gael Clave, Mourad Hadj-Chaib, Laure Leblanc, and Robert de Simone. Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP. In *Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Trento, Italy, June 1999.

[4] Arvind, R.S. Nikhil, D.L. Rosenband, and N. Dave. High-level synthesis: an essential ingredient for designing complex asics. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on, Vol., Iss., 7-11 Nov. 2004*, pages 775–782, 2004.

[5] Pranav Ashar and Sharad Malik. Fast functional simulation using branching programs. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 408–412, San Jose, California, November 1995.

[6] Lennart Beckman, Anders Haraldsson, Östen Oskarsson, and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[7] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[8] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, April 1992. Issue 1652, Mechanized Reasoning and Hardware Design.

[9] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert De Simone. Esterel: A formal method applied to avionic software development. *Science of Computer Programming*, 36(1):5–25, January 2000.

[10] Gérard Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brooks, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, Heidelberg, Germany, 1984.

[11] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[12] Valérie Bertin, Michel Poize, and Jacques Pulou. Une nouvelle méthode de compilation pour le language ESTEREL [A new method for compiling the Esterel language]. In *Proceedings of GRAISyHM-AAA.*, Lille, France, March 1999.

[13] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, August 1993.

[14] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 174–184, Baltimore, Maryland, 1998.

[15] A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. Master's thesis, FIXME, July 1987.

[16] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 236–243, San Jose, California, November 1995.

[17] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A mixed-paradigm simulation/prototyping platform in C++. In *Proceedings of the C++ At Work Conference*, Santa Clara, California, November 1991.

[18] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80, New York, NY, USA, 2007. ACM Press.

[19] Massimiliano Chiodo, Paolo Giusto, Attila Jurecska, Luciano Lavagno, Harry Hsieh, Kei Suzuki, Alberto Sangiovanni-Vincentelli, and Ellen Sentovich. Synthesis of software programs for embedded control applications. In *Proceedings of the 32nd Design Automation Conference*, pages 587–592, San Francisco, California, June 1995. Association for Computing Machinery.

[20] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, volume 2114 of *Lecture Notes in Computer Science*, pages 355–369, Livingston, Scotland, September 2001.

[21] Koen Claessen, Mary Sheeran, and Satnam Singh. Using Lava to design and verify recursive and periodic sorters. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(3):349–358, May 2003.

[22] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 130–145, Minneapolis, Minnesota, 2000.

[23] C. Consel and S. C. Khoo. Semantics-directed generation of a Prolog compiler. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP '91, Passau, Germany, August 1991 (Lecture Notes in Computer Science, vol. 528)*, pages 135–146. Springer-Verlag, 1991.

[24] Charles Consel and Oliver Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 493–501, Charleston, South Carolina, January 1993.

[25] Charles Consel, L. Hornof, Julia L. Lawall, Renaud Marlet, G. Muller, J. Noyé, Scott Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation (SOPE)*, 30(3es), September 1998.

[26] Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[27] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[28] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 554–564, Berlin, Germany, 1996.

[29] Stephen A. Edwards. Compiling Esterel into sequential code. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES)*, pages 147–151, Rome, Italy, May 1999. Association for Computing Machinery.

[30] Stephen A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):169–183, February 2002.

[31] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, April 2003.

[32] Stephen A. Edwards. SHIM: A language for hardware/software integration. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland, April 2005.

[33] Stephen A. Edwards. Using program specialization to speed SystemC fixed-point simulation. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 21–28, Charleston, South Carolina, January 2006.

[34] Stephen A. Edwards, Vimal Kapadia, and Michael Halas. Compiling Esterel into static discrete-event code. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, volume 153(4) of *Electronic Notes in Theoretical Computer Science*, pages 107–121, Barcelona, Spain, March 2004. Elsevier Science.

[35] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, July 2003.

[36] Jeanne Ferrante, Mary Mace, and Barbara Simons. Generating sequential code from parallel code. In *1988 International Conference on Supercomputing*, pages 582–592, St. Malo, France, July 1988. ACM.

[37] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[38] Peter L. Flake, Philip R. Moorby, and G. Musgrave. An algebra for logic strength simulation. In *Proceedings of the 20th Design Automation Conference*, pages 615–618, Miami Beach, Florida, June 1983.

[39] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[40] Arne J. Glenstrup, Henning Makholm, and Jens P. Secher. C-MIX: Specialization of C programs. In *Partial Evaluation—Practice and Theory, DIKU 1998 International Summer School*, pages 108–154, London, UK, 1999. Springer-Verlag.

[41] Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel. [Semantics and models of execution of the synchronous reactive languages: application to Esterel]*. Thèse d'informatique, Université d'Orsay, 1988.

[42] M. Goudarzi, S. Hessabi, and A. Mycroft. Object-oriented asip design and synthesis. In *Forum on Specification and Design Languages (FDL'03)*, Frankfurt, Germany, 2003.

[43] S. Harnett and M. Montenyohl. Towards efficient compilation of a dynamic object-oriented language. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 82–89. Yale, 1992.

[44] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI '99: Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration*, pages 595–619, Deventer, The Netherlands, 2000. Kluwer, B.V.

[45] C. B. Jones, editor. *Lisp as the language for an incremental computer*. The MIT Press, 1964.

[46] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[47] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Upper Saddle River, New Jersey, June 1993.

[48] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In S.L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Springer-Verlag, 1991.

[49] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[50] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[52] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet

Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[53] G. Kildall. A unified approach to global program optimization. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[54] Stephen Cole Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, New Jersey, 1952.

[55] Yanbing Li and Miriam Lesser. HML: An innovative hardware design language and its translation to VHDL. In *Proceedings of the International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, Chiba, Japan, August 1995.

[56] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 317–324, Jersey City, New Jersey, September 2005.

[57] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In *COOTS*, pages 1–20, 1997.

[58] Rajeev Murgai, Fumiyasu Hirose, and Masahiro Fujita. Logic synthesis for a single large look-up table. In *International Workshop on Logic Synthesis*, pages 6–11–6–19, 1995.

[59] André Costi Nácul and Tony Givargis. Code partitioning for synthesis of embedded applications with Phantom. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 190–196, San Jose, California, November 2004.

[60] Hanne Riis Nielson and Flemming Nielson. Bounded fixed point iteration. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 71–82, Albuquerque, New Mexico, 1992.

[61] David A. Penry and David I. August. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference*, pages 926–931, Anaheim, California, June 2003.

[62] Dumitru Potop-Butucaru. *Optimizing for Faster Simulation of Esterel Programs*. PhD thesis, INRIA, Sophia-Antipolis, France, August 2002.

[63] Dumitru Potop-Butucaru. Optimizations for faster execution of Esterel programs. In *Proceedings of the 1st International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 227–236, Mont St. Michel, France, June 2003.

[64] Radu Rugina and Martin Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2017 of *Lecture Notes in Computer Science*, pages 34–48, Yorktown Heights, New York, August 2000.

[65] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4):452–499, 2003.

[66] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. *asp-dac*, 00:238–243, 2004.

[67] Barbara Simons and Jeanne Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR–03.465, IBM, Santa Teresa Laboratory, San Jose, California, February 1993.

[68] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft, October 1993.

[69] Olivier Tardieu and Stephen A. Edwards. R-SHIM: Deterministic concurrency with recursion and shared variables. In *Proceedings of the 4th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, page 202, Napa, California, July 2006.

[70] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, October 2006.

[71] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston, Massachusetts, 1991.

[72] Steven W. K. Tjiang and John L. Hennessy. Sharlit: a tool for building optimizers. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, pages 82–93, New York, New York, 1992.

[73] Manish Vachharajani, Neil Vachharajani, and David I. August. The Liberty structural specification language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, June 2004.

[74] G. A. Venkatesh and Charles N. Fischer. Spare: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 18(4):304–318, 1992.

[75] Reinhard Wilhelm. Program analysis—a toolmaker's perspective. *ACM Computing Surveys*, 28(4es):177, 1996.

[76] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng,

Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[77] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 246–259, Charleston, South Carolina, 1993.

[78] Jia Zeng and Stephen A. Edwards. Separate compilation for synchronous modules. In *Proceedings of the 2nd International Conference on Embedded Software and Systems (ICESS)*, volume 3820 of *Lecture Notes in Computer Science*, pages 129–140, Xi'an, China, December 2005.

[79] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating fast code from concurrent program dependence graphs. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 175–181, Washington, DC, June 2004.

Appendix A

AG Syntax

ag-phase:

> $\texttt{Phase}$ *identifier* (*parameter-list$_{opt}$*) *compound-statement*

parameter-list:

> *parameter*
>
> *parameter-list* , *parameter*

parameter:

> *type identifier*

type:

> *basic-type*
>
> *extensible-class-type*
>
> $\texttt{Set}$ < *type* >
>
> $\texttt{Map}$ < *type* , *type* >

basic-type: one of

> ```
> int  bool  void
> ```

extensible-class-type: one of

> ```
> Alias  Opnd  Instr  Block  Region  Func
> ```

compound-statement:

    { *statements* }

statements:

    statement

    statements statement

statement:

    variable-declaration

    function-definition

    extend-class-definition

    assignment-expression$_{opt}$;

    if-else-statement

    foreach-statement

    phoenix-foreach

    `continue` ;

    `break` ;

    `return` *expression$_{opt}$* ;

    cpp-code-segment

    compound-statement

variable-declaration:

    type variable-declaration-list ;

variable-declaration-list:

    variable

    variable-declaration-list , *variable*

variable:

    identifier

    identifier = expression

function-definition:

       basic-function-definition

       transfer-function-definition

       compute-function-definition

basic-function-definition:

       type identifier (*parameter-list$_{opt}$*) *compound-statement*

transfer-function-definition:

       type `TransFunc` (*direction*) *compound-statement*

compute-function-definition:

       compute-function-name (*identifier$_{opt}$*) *compound-statement*

compute-function-name: one of

       `compose  meet  result`

extend-class-definition:

       `extend class` *extensible-class-type compound-statement*

assignment-expression:

       variable-or-field assignment-operator expression

       expression

variable-or-field:

       variable-or-field –> identifier

       identifier

expression:

> *numeric-literal*
>
> *variable-or-field*
>
> *expression binary-operator expression*
>
> ! *expression*
>
> – *expression*
>
> *variable-or-field* (*variable-list$_{opt}$*)
>
> (*expression*)

variable-list:

> *variable-or-field*
>
> *variable-list* , *variable-or-field*

binary-operator: one of

> + – * < > && || <= >= != ==

assignment-operator: one of

> = += -= *=

if-else-statement:

> if (*expression*) *statement*
>
> if (*expression*) *statement* else *statement*

foreach-statement:

> foreach (*type identifier* in *expression* where$_{opt}$ *direction$_{opt}$*) *compound-statement*

where:

> where *expression*

direction: one of

> forward backward

phoenix-foreach:

  phoenix-foreach-keyword (*parameter-list$_{opt}$*) *compound-statement*

cpp-code-segment:

  /% C++-program-text %/

Appendix B

Recursive FFT Example in SHIM

```
/*********************************
  An in-place complex-to-complex FFT
***********************************/

struct complex{ //structure of complex number
  float real;
  float imag;
};

// main function
// which reads a series of input sample p
// and outputs a series of q after transform.
void main(complex p, complex &q){

  int n = 8;  //sample rate

  stage(p, q, n, 1);

}

// for n input samples,
// log(n) stages will be built recursively,
// each of which is characterized with a different "step" size.
void stage(complex a, complex &b, int n, int step){
  int f;
  int i;
  complex tmp;

  if (step == n){
```

```
    for(i = 0; i < n; i++)
      next b = next a;
    return;
  }

    butterflies(a,tmp,0,0,n,step);
  par
    stage(tmp, b, n, (step*2));

}

void butterfly(complex x, complex y, complex w,
                complex &xx, complex &yy){
  complex t;

  recv x; par recv y;

  t = multiply(w, y);
  yy = minus(x, t);
  xx = plus(x, t);
}

// iteratively constructs butterfly() for each stage
void butterflies(complex a, complex &b,
                  int f, int i, int n, int step){
  complex bb;
  complex b1, b2;
  complex a1, a2;
  complex w;

  if (f >= step)
   return;

  if (i >= n) {
    f += 1;
    i = f;
    butterflies(a,b,f,i,n,step);
    return;
  }

  w.real = cos(f*PI/(2*step));
  w.imag = -sin(f*PI/(2*step));

  {
    for(int j = 0; j < n; j++){
```

```
      recv a;
      if(j == i)
        next a1 = a;
      par
      if(j == (i+step))
        next a2 = a;
    }
  }
  par
    butterfly(a1, a2, w, b1, b2);
  par
    butterflies(a,bb,f,(i+2*step),n,step);
  par
  {
    for(int j = 0; j < n; j++){ //synchronize output
      recv bb;
      if(j == i)
        next b = next b1;
      else if(j == (i+step))
        next b = next b2;
      else
        next b = bb;
    }
  }

}

//facility functions

float sin(float x){
  float x3,x5,x7;

  if(x > PI/2) x = PI - x; //PI = 3.14

  x3 = x^3/(2 * 3);
  x5 = (x3 * x^2)/(4 * 5);
  x7 = (x5 * x^2)/(6 * 7);

  return (x - x3 + x5 - x7);
}

float cos(float x){
  return sin(PI/2 - x);
}
```

```
complex plus(complex x, complex y){
  complex z;
  z.real = x.real + y.real;
  z.imag = x.imag + y.imag;
  return z;
}

complex minus(complex x, complex y){
  complex z;
  z.real = x.real - y.real;
  z.imag = x.imag - y.imag;
  return z;
}

complex multiply(complex x, complex y){
  complex z;
  z.real = x.real * y.real - x.imag * y.imag;
  z.imag = x.real * y.imag + x.imag * y.real;
  return z;
}
```