Efficient, Deterministic and Deadlock-free Concurrency

Nalini Vasudevan

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2011

ABSTRACT

Efficient, Deterministic and Deadlock-free Concurrency

Nalini Vasudevan

Concurrent programming languages are growing in importance with the advent of multicore systems. Two major concerns in any concurrent program are data races and deadlocks. Each are potentially subtle bugs that can be caused by non-deterministic scheduling choices in most concurrent formalisms. Unfortunately, traditional race and deadlock detection techniques fail on both large programs, and small programs with complex behaviors.

We believe the solution is model-based design, where the programmer is presented with a constrained higher-level language that prevents certain unwanted behavior. We present the SHIM model that guarantees the absence of data races by eschewing shared memory.

This dissertation provides SHIM based techniques that aid determinism - models that guarantee determinism, compilers that generate deterministic code and libraries that provide deterministic constructs. Additionally, we avoid deadlocks, a consequence of improper synchronization. A SHIM program may deadlock if it violates a communication protocol. We provide efficient techniques for detecting and deterministically breaking deadlocks in programs that use the SHIM model.

We evaluate the efficiency of our techniques with a set of benchmarks. We have also extended our ideas to other languages. The ultimate goal is to provide deterministic deadlock-free concurrency along with efficiency. Our hope is that these ideas will be used in the future while designing complex concurrent systems.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Here is my consolidated list of people I would like to thank:

- Stephen Edwards (Columbia University)
  I would like to foremost thank my advisor, who did a perfect mix of micro and macro managing and at perfect times. Without his constant encouragement, I would not have been able to start and complete my thesis. I owe a major part of my success to Stephen.

- Alfred Aho, Luca Carloni, Martha Kim (Columbia University)
  I can best describe them as 'easily approachable'. Not only did they serve on my thesis committee, but they also never denied when I asked for recommendations.

- Julian Dolby, Vijay Saraswat, Olivier Tardieu (IBM Research)
  A major part of my ideas towards my thesis originated at IBM. I would like to thank them for their guidance and providing me with the right infrastructure.

- Kedar Namjoshi (Bell Laboratories)
  I would like to thank Kedar for his expertise and guidance in theoretical concepts without which I would have felt incomplete.

- Satnam Singh (Microsoft Research)
  The biggest stepping stone of my thesis was my first paper with Satnam. I would like to thank him for sponsoring me a paid vacation to the United Kingdom and also for serving on my thesis.

- Prakash Shankor, Saraswathi Vasudevan, Vasudevan Venkataraman (My family)
  I blame Prakash for igniting the idea of PhD in my head and letting me do what I wanted to. I owe the entire non-technical part of my thesis to him. I finally thank my parents for letting me fulfil their wishes.

Part I

Introduction

Chapter 1

The Problem

Multicore shared-memory multiprocessors now rule the server market. While such architectures provide better performance per watt, they present many challenges.

Scheduling—instruction ordering—is the biggest issue in programming shared-memory multiprocessors. While uniprocessors go to extremes to provide a sequential execution model despite caches, pipelines, and out-of-order dispatch units, multiprocessors typically only provide such a guarantee for each core in isolation; instructions are at best partially ordered across core boundaries.

Controlling the scheduling on multiprocessors is crucial not only for performance, but because data races can cause scheduling choices to change a program's function. Worse, the operating system schedules nondeterministically.

We say that a program produces nondeterministic output if it is capable of producing different outputs during reruns of the program with the same input. The program in Figure 1 is nondeterministic. It uses C++-like semantics with Cilk [19]-like constructs for concurrency. It creates two tasks f and g in parallel using the *spawn* construct. Both functions take x by reference. Clearly, x is getting modified concurrently by both the tasks, so the value printed by this program is either 3 or 5 depending on the schedule. One way to avoid races is to protect x by a lock and thereby ensure atomic updates to x, but this still gives nondeterministic output. This is because operations within atomic blocks are not commutative.

Such nondeterministic functional behavior arising from timing variability–a data race–is among the nastiest things a programmer may confront. It makes debugging all but impossible because unwanted behavior is rarely reproducible [11]. Rerunning a nondeterministic program on the same input usually does not produce the same behavior. Inserting assert or print statements or running the program in a debugger usually changes timing enough to make the bug disappear. Debugging such programs is like trying to catch a butterfly that is only visible from the corner

```
void f(int &a) {
  a = 3;
}

void g(int &b) {
  b = 5;
}

main() {
  int x = 1;
  spawn f(x)
  spawn g(x);
  sync; /* Wait for f and g to finish */
  print x;
}
```

Figure 1.1: A nondeterministic parallel program

of your eye.

We believe a programming environment should always provide functional determinism because it is highly desirable and is very difficult to check for on a per-program basis. Virtually all sequential programming languages (e.g., C) are deterministic: they produce the same output given the same input. Inputs include usual things such as files and command-line arguments, but for reproducibility and portability, things such as the processor architecture, the compiler, and even the operating system are not considered inputs. This helps programmers by making it simpler to reason about a program and it also simplifies verification because if a program produces the desired result for an input during testing, it will do so reliably.

By contrast, concurrent software languages based on the traditional shared memory, locks, and condition variables model (e.g., pthreads or Java) are not deterministic by this definition because the output of a program may depend on such things as the operating system's scheduling policy, the relative execution rates of parallel processors, and other things outside the application programmer's control. Not only does this demand a programmer consider the effects of these things when designing the program, it also means testing can only say a program *may* behave correctly on certain inputs, not that it will.

A few concurrent programming languages provide determinism through semantics. SHIM [47; 116] is one such instance. It is an asynchronous concurrent language that is scheduling independent: its input/output behavior is not affected by any nondeterministic scheduling choices taken by its runtime environment due to processor speed, the operating system, scheduling policy, etc. A SHIM program is composed of sequential tasks that synchronize whenever they want to communicate. The language is a subset of Kahn networks [70] (to ensure determinism) that employs the rendezvous of Hoare's CSP [62] for communication to keep its behavior tractable.

Kahn's unbounded buffers would make the language Turing complete, even with only finite-state processes, so the restriction to rendezvous makes the language easy to analyze. Furthermore, since SHIM is a strict subset of Kahn networks, it inherits Kahn's scheduling independence: the sequence of data values passed across each communication channel is guaranteed to be the same for all correct executions of the program (and potentially input dependent).

The central hypothesis of SHIM is that deterministic concurrent languages are desirable and practical. That they relieve the programmer from considering different execution orders is clear; whether they impose too many constraints on the algorithms they can express is also something we attempt to answer here.

Although SHIM is deterministic, it is not deadlock free; a programmer may use language constructs incorrectly to cause the program to deadlock. We demonstrate that deadlocks can be easily detected statically because of the deterministic property of SHIM.

Our ultimate goal is to have both determinism as well as deadlock freedom. In the next section, we discuss the terms used in this thesis followed by the problem we are addressing. Then, we discuss some of things to remember while designing deterministic, deadlock free systems. Finally, we give an outline of this thesis.

1.1 Terminology

This section defines the terms used in this thesis.

A *multi-core processor* is a system that consists of two or more cores. Multicores are used for reduced power consumption and simultaneous processing of multiple tasks, therefore resulting in enhanced performance. A *task* or a *process* is a sequential unit of computation. A *sequential program* has a single task. By contrast, a *parallel program* consists of multiple tasks that may execute concurrently.

A *programming model* is an abstraction or a template to express algorithms. *Programming languages* are more concrete and are based on programming models. They have specific forms of syntax and semantics.

An *application* is an instance of a sequential or parallel program that implements an algorithm in a programming language. A *benchmark* is a set of standard applications used to assess the performance of something, usually by running a number of standard tests.

1.2 Problem Statement

This thesis wishes to provide programming language support for the following:

- **Determinism**: By determinism, we mean that the output of a program depends only on the input of the program and not on the running environment. Inputs include things such as files and command-line arguments. We do not deal with reactive systems. Programming environment includes things such as the processor architecture, compiler, and even the operating system and these are not considered inputs.

- **Deadlock-freedom**: A deadlock is a situation in which two or more tasks wait for each other to make progress, but neither ever does causing an indefinite wait. A deadlock usually arises because of improper synchronization. We require techniques to detect and avoid these situations. We do not wish to solve the termination problem.

1.3 Design Considerations

While we design a deterministic and a deadlock-free system, our goal is to achieve three things: performance, scalability and programmer flexibility.

1.3.1 Performance

A general hypothesis is that determinism introduces performance degradation because of synchronization. There are two types of synchronization: centralized and distributed. A centralized synchronization forces all tasks in a system to synchronize while a distributed synchronization forces only a subset of tasks to synchronize. Distributed methods perform better because the tasks have to wait less, but they are more susceptible to deadlocks. An out-of-order synchronization between subsets of tasks may lead to a deadlock. On the other hand, in centralized systems, deadlocks are avoided because all tasks are forced to synchronize at the same point.

In some cases, the programming environments are nondeterministic, but there are techniques and tools to check for determinism and deadlocks during runtime. The problem with these tools is that they add a considerable amount of overhead that reduces performance drastically.

Part	Chapter	Question to answer	Published
	3	How can we achieve determinism ?	SES 2010 [50]
II	4	Is determinism efficient?	DATE 2008 [51]
(Determinism)	5	Is determinism practical?	SAC 2009 [124]
	6	Determinism: Language vs. Library ?	IPDPS 2008 [130]
III	7	How do we solve the deadlock problem?	MEMOCODE 2008 [122]
(Deadlock-freedom)	8	How can we efficiently detect deadlocks?	EMSOFT 2009 [109]
	9	How can we deterministically break deadlocks?	HOTPAR 2010 [127]
	10	How can we enforce deadlock freedom?	HIPC-SRS 2010 [128]
IV	11	Can we reduce memory in deterministic	MEMOCODE 2009 [123]
(Efficiency)		programs?	TCAD 2010 [126]
	12	Can we optimize deterministic constructs?	CC 2009 [131]
	13	Can we optimize locks?	PACT 2010 [129]
V	14	What are the limitations?	IPDPS Forum 2008 [125]
(Conclusions)		What next?	PLDI-FIT 2009 [125]

Table 1.1: Thesis outline

1.3.2 Scalability

A number of programming environments provide determinism at compile time. Static verifiers and type systems are examples of such environments. These techniques do not explicitly introduce deadlocks but they do not scale at compile time because they have to consider all possible interleavings of tasks in the program.

Among the systems that provide determinism at runtime, distributed systems are known to scale better than centralized systems in both performance and ease of implementation.

1.3.3 Programmer Flexibility and Ease of Use

Most deterministic programming models provide determinism by imposing a number of restrictions. Most type systems require programmers to explicitly annotate the program. Static verifiers do not force any restrictions on the program, but they simply do not scale with flexible programs and give false positives as results. Our goal is to achieve a balance between performance, scalability and programmer flexibility.

1.4 Thesis Outline

Table 1.1 gives the overview of this thesis. We first provide a background study in Chapter 2. We then begin by describing the SHIM model in Part II. We evaluate our model by generating code for different architectures. We illustrate a backend in Chapter 4 for SHIM that generates C code that made calls to the POSIX thread (pthread) library to ask for parallelism. Each communication action acquires the lock on a channel and checks whether every process connected to it also had blocked (i.e., whether the rendezvous could occur).

We also illustrate a backend for IBM's CELL processor in Chapter 5. A direct offshoot of the pthread backend, it allows the user to assign computationally intensive tasks to the CELL's synergistic processing units (SPUs); remaining tasks run on the CELL's PowerPC core (PPU).

Next, we illustrate the feasibility of SHIM as a library. We provide a deterministic concurrent communication library in Chapter 6 for an existing multithreaded language. We implemented the SHIM in the Haskell functional language, which supports transactional memory.

SHIM is interesting because it is deterministic but it is not deadlock free. We provide simple techniques to detect deadlocks in SHIM in Part III. SHIM does not need to be analyzed under an interleaved model of concurrency since most properties, including deadlock, are preserved across schedules. In Chapter 7, we use a synchronous model checker NuSMV [34] to detect deadlocks in SHIM—a surprising choice since SHIM's concurrency model is fundamentally asynchronous. We later take a compositional approach in Chapter 8 in which we build an automaton for a complete system piece by piece. The result: our explicit model checker outperforms the implicit NuSMV on these problems. Our evaluations led to other directions. We wanted a more robust concurrent programming model that is both deterministic and deadlock free – we discuss the D^2C model in Chapter 10.

We then provide a few optimization techniques to improve the efficiency of SHIM and other related languages like X10 [29] in Part IV. To improve the efficiency of the SHIM model, we applied model checking to search for situations where buffer memory can be shared [123; 126]. In general, each communication channel needs its own space to store any data being communicated over it. However, in certain cases, it is possible to prove that two channels can never be active simultaneously and thus share buffer memory.

In Chapter 12, we describe a tool that mitigates the overhead of general-purpose clocks in IBM's X10 language by analyzing how programs use the clocks and then by choosing optimized implementations when available. These clocks are deterministic barriers and are similar to SHIM's communication constructs.

The major bottleneck of deterministic programs is due to synchronization. Synchronization constructs are implemented using low level locks. In Chapter 13, we describe efficient locking algorithms for specialized locking behavior.

Finally, we discuss the limitations of SHIM in Chapter 14. We report our conclusions and open new directions for future work.

Chapter 2

Background

Concurrent programming languages suffer from a number of problems including nondeterminism and deadlocks. This chapter surveys the various issues involved while designing concurrent systems and particularly focuses on techniques that deal with nondeterminism and deadlocks. We provide a survey of the various programming models, tools and techniques that are in use today to build concurrent systems, and specifically how they address deadlock and nondeterminism problems at various levels — compiler, programming language, operating systems and hardware.

2.1 Problems with concurrent programming

Concurrency comes with an abundant number of problems. We list a few below.

- **Paradigm shift:** Sequential computers were ruling the world but not anymore. Most programmers find concurrency hard because they are trained to think sequentially.

- **Lack of a good model:** There is no widely accepted concurrent programming or memory model. The next section surveys the programming models that are in use today and discusses their pros and cons.

- **Concurrency bugs:** Bugs like nondeterminism and deadlocks that are virtually absent in sequential programming are exposed in concurrent programming. We list some of the concurrency bugs here:

  - *Nondeterminism:* A condition when some possible interleaving of tasks results in undesired program output.

  - *Deadlock:* A state in which two or more tasks indefinitely wait for each other.
  - *No Fairness:* A condition when some task does not get a fair turn to make progress.
  - *Starvation:* A state when a task is deprived of a shared resource forever.

- **Portability:** Programmers are generally required to have knowledge about the underlying layers(no. of cores, operating system scheduling policy, cache size and policy, memory layout and other hardware features) to produce efficient concurrent programs. Therefore, a program written for one architecture may not be suitable for another architecture resulting in poor portability. Also, with emerging and changing architectures, programs may have to be rewritten to suit different architectures.

This thesis mainly addresses nondeterminism and deadlocks, although we believe that all the issues listed are equally important. We also try not to neglect these issues while designing deterministic and deadlock-free systems.

2.2 Concurrent programming models

Concurrent programming models are becoming more prominent with the advent of multicore systems. They provide a layer of abstraction between the application and the underlying architecture including the operating system. A programming model may choose to hide or expose aspects of the operating system and hardware. Specifically, a concurrent programming model controls the concurrency features provided by the operating system or hardware.

Generally, the more the model exposes, the more efficient code can a programmer write. As a consequence of more exposure, the programmer has to explicitly work with the lower layers and therefore, productivity is reduced. He is also exposed to a number of bugs like nondeterminism and deadlocks, since he has access to lower layers.

Alternatively, a programming model may choose to expose very little of the underlying layers and hence release the programmers the burden of dealing with low level details that include the operating system and hardware. Such a model may also hide low level bugs, thereby allowing programmers to deal with bug-free code. The SHIM model is an instance of this kind of programming model. It abstracts away nondeterminism from the programmer.

The SHIM model forces synchronization of tasks while accessing shared data to provide determinism. The model eliminates data races by design and also simplifies the deadlock detection process. Tasks in SHIM can be created in parallel using the *par* statement. It uses message-passing-like semantics for communication. We discuss the model in detail in the next chapter.

Message passing is a well known approach used by parallel tasks to communicate with each other and works well for distributed systems. The Message Passing Interface (MPI) is a popular standard library for creating and communicating between concurrent threads. The communication pattern is flexible (blocking, unblocking, variable buffer size) and easily programmable. MPI was not designed to deal with issues such as nondeterminism and deadlocks; the programmer has to deal with these issues explicitly.

CSP (Communicating Sequential Processes)[61] is another parallel programming model that uses message passing. The communication is blocking – both the sender and the receiver have to rendezvous for the communication to be successful. A task may choose to wait on two or more channels at the same time, and resume execution as soon as data is available on one of the channels. This makes the output dependent on time, making the model nondeterministic.

By contrast, a Kahn network is a deterministic concurrent programming model that uses message passing for communication. A Kahn Network [70] is composed of a set of communication processes that may send and receive on channels. Each communication channel connects a single sending process with a single receiving process. The communication structure of a system is therefore a directed graph whose nodes are processes and whose arcs are channels. There is no shared data; processes communicate only through channels. The receiver process is blocking: it waits until the sender writes the data. The receiver cannot choose to wait based on whether the data is available or not. This property makes the model deterministic. The sender is nonblocking; it writes to one end of the channel and the receiver reads from the other end. The channel is implemented as an unbounded buffer.

Figure 2.1 is an example of a Kahn processes and its corresponding network is shown in Figure 2.2. f, g and h are three parallel tasks created by the *par* construct in *main*(). The two producer tasks f and g send values 1 (on channel a) and 0 (on channel b) respectively. Task h receives the values from channels a and b into variable j. j sees an alternating stream of 1's and 0's.

In Figure 2.1, suppose f runs faster than g or h, then the channel a fills in quickly. However, h will not be able to receive the data as quickly as f sends. Therefore, there will be an accumulation of data on the channel. This is not a problem in Kahn's model, because the channel acts as an infinite queue between the producer and the consumer.

In practice, this infinite bound is impossible to implement. The SHIM model provides functional determinism by adopting Kahn networks, and also solves the unbounded buffer problem by using CSP-style rendezvous for communication. The sender and the receiver have to wait for each other to communicate data.

```
void f(out a)
{
   for(;;) {
      send a = 1; /* sends 1 on channel a */
   }
}

void g(out b)
{
  for(;;) {
      send b = 0; /* sends 0 on channel b */
  }
}

void h(in a, in b) {
  int j;
   for (int i = 0; i++; ) {
     if (i%2)
        j = recv a; /* receives 1 */
     else
        j = recv b; /* receives 0 */
  }
}

main() {
   chan int a, b;
   f(a) par g(b) par h(a, b); /* Runs the three tasks in parallel */
}
```

Figure 2.1: Example of Kahn Processes

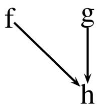


Figure 2.2: Kahn network of Figure 2.1

Concurrent programming models also control the mode of parallelism and it can be broadly classified into two types: data level parallelism and task level parallelism. Data level parallelism forces parts of data to be distributed over different processors and computed concurrently. A classic example is allowing different elements of an array to be processed concurrently. By contrast, task level parallelism is allowing code to run concurrently. For instance, SHIM supports task level parallelism.

Programming models also restrict the class of applications that can be implemented. MIT's StreamIt [119] model, for example, is primarily suitable for stream processing applications. Stream computing has various applications including image and single processing. It is based on synchronous data flow [76] that operates on streams of data known as tokens. These tokens pass through a number of computation units known as filters. Filters communicate with each other through channels. Channels are implemented as buffers and pass tokens. StreamIt programs have single input and single output filters. Filters use *push*, *pop* and *peep* functions to operate on input and output streams. Streams can be pipelined. They can also be split and joined for data level parallelism.

StreamIt is completely deterministic. It has simple static verification techniques for deadlock and buffer overflow. However, StreamIt is a strict subset of SHIM and StreamIt's design limits it to a smaller class of applications.

By contrast, Cilk [19] is a an interesting programming language that it covers a larger class of applications. It is C based and the programmer must explicitly ask for parallelism using the *spawn* and the *sync* constructs. Cilk is definitely more expressive than SHIM and StreamIt. However, Cilk allows data races. Figure 1, for example, is a nondeterministic concurrent program in Cilk. Explicit techniques [30] are required for checking data races in Cilk programs.

X10 [29; 106] is another language that adopts the Cilk model. It uses *async* and *finish* instead of *spawn* and *sync*. It is a parallel and distributed object-oriented language. To a Java-like sequential core it adds constructs for concurrency and distribution through the concepts of *activities* and *places*. An activity is a unit of work, like a thread in Java, and is created by an *async* statement; a place is a logical entity that contains both activities and data objects.

Just like Cilk, the X10 language allows races and does not impose hard restrictions on how activities should be created. We describe the language in detail in Chapter 12.

Synchronous programming languages like Esterel [17] are deterministic. An Esterel program executes in clock steps and the outputs are synchronous with its inputs. Although an Esterel program is susceptible to causality problems, this form of deadlock can be detected at compile time. Unfortunately, synchronous models require constant, global synchronization and force designers to explicitly schedule virtually every operation. Although standard in hardware designs, global synchronization is costly in software. Furthermore, the presence of a single global clock effectively forces entire systems to operate at the same rate. Frustration with this restriction was one of the original motivations for SHIM.

2.3 Determinizing tools

A number of tools provide determinism. For example, Kendo is a software system that deterministically multithreads concurrent applications. Kendo [93] ensures a deterministic order of all lock acquisitions for a given program input. Consider two threads, T_1 and T_2 in Figure 2.3. Suppose x is initialized to 0, then the final value of x is either 1 or 2, depending on which thread acquires the lock first. Kendo removes this nondeterministic behavior by deterministically ordering the acquisition of locks. An example of deterministic ordering is lowest thread id first. In this case, Kendo waits for all threads to contend for the lock, then forces T_1 to acquire the lock before T_2, thereby always giving the final value of x as 2.

```
 Thread T1      Thread T2
lock(m);       lock(m);
 x++;           x*=2;
unlock(m);     unlock(m);
```

Figure 2.3: Two threads T_1 and T_2 running in parallel

Kendo comes with three shortcomings. It operates completely at runtime, and there is a considerable performance penalty. Secondly, if we have the sequence *lock(A); lock (B)* in one thread and *lock(B); lock(A)* in another thread, a deterministic ordering of locks may still deadlock. Thirdly, the tool operates only when shared data is protected by locks.

Software Transactional Memory (STM) [110] is an alternative to locks: a thread completes modifications to shared memory without regard for what other threads might be doing. At the end of the transaction, it checks for conflict freedom and commits if the validation was successful, otherwise it rolls back and re-executes the transaction. STM mechanisms avoid races but do not solve the nondeterminism problem.

Berger's Grace[16] is a runtime tool that is based on STM. If there is a conflict during commit, the threads are committed in a particular sequential order (determined by the order in which they appear in the source code), ensuring determinism. For instance, for the code in Figure 1, f commits before g, therefore resulting in output value 5. Grace works on Cilk programs. The tool ensures that the output of the concurrent code is same as its sequential equivalent, and this sequential equivalent is obtained by removing *spawn* and *sync* statements from the concurrent program.

The problem with Grace is that it incurs a lot of runtime overhead. This dissertation partially solves this overhead problem by addressing the issue at compile time and thereby reducing a considerable amount of runtime overhead.

Like Grace, Determinator[6] is another tool that allows parallel processes to execute as long as they do not share resources. If they do share resources and the accesses are unsafe, then the operating throws an exception (a page fault).

Cored-Det [15], based on DMP [42] uses a deterministic token that is passed among all threads. A thread to modify a shared variable must first wait for the token and for all threads to block on that token. DMP is hardware based. Although, deadlocks may be avoided, we believe this setting is nondistributed because it forces all threads to synchronize and therefore leads to a considerable performance penalty. In the SHIM setting, only threads that share a particular channel must synchronize on that channel; other threads can run independently.

Deterministic replay systems [31; 5] facilitate debugging of concurrent programs to produce repeatable behavior. They are based on record/replay systems. The system replays a specific behavior (such as thread interleaving) of a concurrent program based on records. The primary purpose of replay systems is debugging; they do not guarantee determinism. They incur a high runtime overhead and are input dependent. For every new input, a new set of records is generally maintained.

Like replay systems, Burmin and Sen [23] provide a framework for checking determinism for multithreaded programs. Their tool does not introduce deadlocks, but their tool does not guarantee determinism because it is merely a testing tool that checks the execution trace with previously executed traces to see if the values match. Our goal is to guarantee determinism at compile time – given a program, it will generate the same output for a given input.

2.4 Model checkers and verifiers

There are a number of model checkers that verify concurrent programs. SPIN [63], for instance, supports modeling of asynchronous processes. Properties to be verified are given as Linear Temporal Logic (LTL). SPIN expands all possible interleavings to verify a concurrent program. It is a general purpose tool and can be used to verify concurrent programs for properties including determinism and deadlocks. The problem with model checkers is that they do not scale to large programs. Also, they cannot express programs with complex structures and behaviors.

Martin Vechev's tool [132] finds determinacy bugs in loops that run parallel bodies. It analyzes array references and indices to ensure that there are no read-write and write-write conflicts.

Type and effect systems like DPJ [20] have been designed for deterministic parallel programming. These systems do not themselves introduce deadlocks, but type systems generally require programmer annotations. SHIM does not require annotations; it provides restrictions through its constructs. One may argue against learning a new programming paradigm or language like SHIM, but SHIM can be implemented as a library (Chapter 6) and the deadlock detector (Part III) can be incorporated into it. The second problem with annotation based systems is that the programmer has to ensure correct annotation; otherwise it results in incorrect effect propagations.

Part II

Determinism

Outline

This part illustrates techniques to guarantee input-output determinism. We use a combination of compile-time and runtime techniques to obtain scheduling-independent behavior. Our approach is a deterministic programming model and language – SHIM. We start by explaining SHIM and its semantics. We then provide ways to generate efficient runtime code from SHIM programs for different architectures. We finally provide a deterministic concurrent library in Haskell that adopts the SHIM model for race-free behavior.

Chapter 3

The SHIM Model

Because of the popularity of multicore chips, there is a growing need for programming techniques, models, and languages that help exploit parallel hardware. In this chapter, we describe the concurrency model underlying a programming language called SHIM—"software/hardware integration medium" [116] for its initial bias toward embedded system applications–to ease the transition from single-threaded software to robust multicore-aware implementations.

One of the key features of SHIM is that the output behavior of a program is deterministic: the output of a program just depends on its input; it does not depend on the environment such as compiler, runtime, OS, or hardware platform. Concurrent tasks in SHIM run asynchronously and do not share any data. The environment may schedule the tasks in any way (i.e., different schedules produce different interleavings of the tasks), but still the program will produce deterministic output. If the tasks have to share data, they have to synchronize using rendezvous communication, and the SHIM's runtime system takes care of this. By rendezvous, we mean that all tasks sharing a particular variable have to meet – in a way similar to a barrier.

The deterministic property of SHIM simplifies validation. Most programs are still validated by simply running them. It is hard enough to validate a deterministic, sequential program with such an approach: the user must create an appropriate set of test cases and check the results of running the program on these cases. If the model is nondeterministic, as with most concurrent formalisms, even running a program on a test case only tells us what the result of running the program might be. It does not guarantee that the result is correct. A different testing environment may cause the program to behave differently for the same input. This is not the case with SHIM because it guarantees scheduling independence.

The SHIM model and language [47; 115] prevent data races by providing scheduling independence: given the same input, a program will produce the same output regardless of what scheduling choices its runtime environment makes. It provides certain program correctness guarantees and makes others easy to check by adopting CSP's rendezvous [62] in a Kahn network [70] setting. In particular, SHIM's scheduling independence makes other properties easier to check because they do not have to be tested across all schedules; one is enough. Deadlock is one such property: for a particular input, a program will either always or never deadlock; scheduling choices (i.e., different interleaving of tasks) cannot cause or prevent a deadlock.

SHIM [116] is a C-like language with additional constructs for communication and concurrency. Specifically, *p par q* runs statements *p* and *q* in parallel, waiting for both to terminate before proceeding; *send c* and *recv c* are blocking communication operators that synchronize on a variable (or a channel) *c*. As an alternative to *send* and *recv*, *next c* is a blocking communication operator that synchronizes on channel *c* and either sends or receives data depending on which side of an assignment (=) the *next* appears.

SHIM tasks communicate exclusively through this multiway rendezvous; there are no global variables or pointers. Any variable that is shared should be a channel and be declared as *chan*. We illustrate SHIM with examples taken from Tardieu's paper [117].

```
void f(chan int a) { // a is a copy of c
  a = 3;
  recv a; // synchronize with g; a gets c's value
              // a = 5
}
void g(chan int &b) { // b is an alias for c
  b = 5;
  send b; // synchronize with f
              // b = 5
}
void main() {
  chan int c = 0;
  f(c); par g(c);
}
```

Here, the program runs two tasks *f* and *g* concurrently. *a* and *b* are incarnations of channel *c*. In *f*, *a* is a copy of *c* that is first modified by `a=3` before being updated by *recv*. By contrast, *b* is an alias for *c* in *g*, so the assignment `b=5` actually modifies *c*. Because they are associated with the same variable, the *send* and *recv* operations must synchronize to execute. When they do, the *recv* statement copies the master value of *a*—*c*, which was set to 5 in *g*—to the local copy of *f*. Thus *a* is 5 just before *f* terminates.

The *next* operation can also used for communication. For instance, in the piece of code we just saw, *recv a* can be replaced by *next a*, and *b = 5; send b;*, can be replaced by *next b = 5*. In other words, *next* behaves like a *send* if it appears on the left side of an assignment, and like *recv* otherwise.

Only the procedure that takes a channel by reference may send on that channel. A channel may be passed by reference to at most one of the procedures. E.g.,

```
int f(chan int &x, chan int y) { x++; y--; }
int g(chan int z) { z--; }
void main() {
  chan int a; a=0; chan int b; b=1;
  f(a, b); par f(b, a); par g(a); // OK: a=1, b=2
}
```

In the above piece of code, executes $f(a,b)$ in parallel with $f(b,a)$, and both run in parallel with $g(a)$; The first f takes a by reference and a is incremented once, while the second f takes b by reference and increments b by 1. g does not take any variable by reference. Therefore, it does not affect the values of a and b. So, the values of a and b become 1 and 2 respectively after the execution of the second line in *main*.

The following line would be illegal in main.

```
  f(a, a); par f(a, b); // incorrect: a is passed twice by reference – compiler reject
```

Due to this restriction (enforced at compile time), concurrently running procedures never share memory - every task maintains its own local copy. The sender task alone references the actual copy, and there can be only one sender task on a channel.

In general, if there are two sender tasks on a particular channel in the code section of the program, then the compiler rejects the program to guarantee determinism. Summarily, the asynchronous parts in SHIM are totally independent because they never share memory. Sharing is only through explicit synchronization using rendezvous communication. This makes SHIM deterministic.

It is not necessary for the statements in the *par* statement to be procedure calls. For instance:

```
void main()
{
 chan int a, b;

 { // Task 1
   a = 5;
   send a; // Send 5 on a (wait for task 2)
   // now a = 5
   recv b; // Receive b (wait for task 2)
   // now b = 10

 } par { // Task 2
   recv a; // Receive a (wait for task 1)
   // now a = 5
```

```
    b = 10
    send b;  // Send 10 on b (wait for task 1)
    // now b = 10
   }
}
```

The SHIM compiler dismantles the above code as:

```
/* Task 1 */
void main1(int &a, int b) {
    a = 5;
    send a;  // Send 5 on a (wait for task 2)
    // now a = 5
    recv b;  // Receive b (wait for task 2)
    // now b = 10
}
/* Task 2 */
void main2(int a, int &b) {
    recv a;  // Receive a (wait for task 1)
    // now a = 5
    b = 10
    send b;  // Send 10 on b (wait for task 1)
    // now b = 10
}

void main()
{
 chan int a, b;
 main1(a,b); par main2(a,b);
 /* a = 5, b = 10 */
}
```

Task 1 (represented by *main*1), being the sender on *a*, takes *a* by reference. Similarly, *main*2 takes *b* by reference. The two peer tasks communicate on channels *a* and *b*. Tasks 1 and 2 are executed in parallel. The *send a* in task 1 waits for task 2 to receive the value. The tasks therefore rendezvous, then continue to run after the communication takes place. Next, the two tasks rendezvous at *b*. This time, task 2 sends and task 1 receives.

Here is another example that illustrates how the *send* and *recv* instructions enable communication between concurrently running procedures.

```
void f(chan int &x) { /* reference to a */
    x = 3; /* modifies a, a is 3 */
    send x; /* sends 3 */
    x = 4; /* modifies a, a is 4 */
 }
void g(chan int y, chan int &z) {
    y = 5; /* modifies local copy */
    recv y; /* receives 3, y is 3 */
    z = y; /* modifies b */
}
```

```
void main() {
  chan int a; a=0; chan int b; b=1;
  f(a); par g(a, b); // a=4, b=3
}
```

Here, *send x* in *f* and *recv y* in *g* synchronize and the value of *x* in *f* is copied into *y* in *g*. Variables *x* and *y* are paired in this communication because both are *instances* of variable *a* from *main*, that is, *x* is a reference to *a* and *y* is a "by-value" reference to *a*. We say that procedures *f* and *g* *share* variable *a* from *main* even if only *f* has access to the value of variable *a* through *x*.

When two or more concurrent procedures share the same variable *a* in this sense, all of them must participate in each communication on *a*. Hence, each procedure reaching a *send x* or *recv y* instruction (where *x* resp. *y* is the name of the local instance of *a*) blocks, that is, stops executing until every procedure involved is blocked on *a*. Then, a communication takes place atomically.

In other words, the primitive communication mechanism in SHIM is the multiway rendezvous that requires all participants in a communication to synchronize – there can be multiple receivers but only one sender on a channel. Of course, other traditional communication mechanisms can be built using this multiway rendezvous. For instance, the fifo procedure described later in this section implements buffered channels.

In SHIM, there are rules for the (static) disambiguation of multiple-sender-multiple-receiver communications. In particular, a procedure can only send values on a pass-by-reference parameter channel. For instance,

```
void snd(chan int &x) { send x; }
void rcv(chan int y) { recv y; }
void main() { chan int a; a=0; snd(a); } // OK
void main() { chan int a; a=0; snd(a); par rcv(a); par rcv(a); } // OK
void main() { chan int a; a=0; snd(a); par snd(a); } // incorrect
void main() { chan int a; a=0; rcv(a); } // OK – receives 0 (the last value on the channel)
void main() { chan int a; a=0; rcv(a); par rcv(a); } // OK – both receive 0
```

In the absence of a sender, the rendezvous deadlocks. Competing synchronization barriers may also cause deadlocks. For example,

```
void f(chan int &x, chan int &y) { send x; send y; }
void g(chan int x, chan int y) { recv x; recv y; }
void main() { chan int a; a=0; chan int b; b=0; f(a, b); par g(b, a); }
// deadlocks
```

Procedures *f* and *g* share *a* and *b* from *main*; *f* is waiting to synchronize on *a* whereas *g* is blocked on *b*. Therefore, neither synchronization attempt completes.

This means coding in SHIM involves tracking down deadlocks, but we prefer reproducible fatal errors to hard-to-detect, hard-to-reproduce data races. Deadlock detection techniques are discussed in the later chapters.

A terminated procedure is no longer compelled to rendezvous. E.g.,

```
void f(chan int &x, chan int &y) { send x; send y; send x; }
void g(chan int x) { recv x; }
void main() {
  chan int a; a=0; chan int b; b=0;
  f(a, b); par g(b); // no deadlock: a is only shared by f
  f(a, b); par g(a); // no deadlock: a is only shared by f once g returns
}
```

This is one of the two reasons multiway rendezvous is fundamental to SHIM. Because procedures may terminate, a multiway channel may dynamically shrink; because concurrent procedures may further divide into more concurrent procedures, a multiway channel may dynamically extend. A procedure (or a task) that takes a channel by value, may pass the channel to its subprocedures (or subtasks) only by value.

Summarily, a *send x* or a *recv x* waits for all tasks that access channel x, to either communicate on x or terminate. Once this condition is satisfied, the value is copied from sender to all receivers. If there is no sender at the rendezvous, the last value written on the channel is copied to the receivers. After this, the tasks continue execution independently. When a task executes a statement $x = a$, it writes a to its local copy of x if the task is a receiver. The sender alone writes to the actual location of x.

To perform I/O in SHIM, we declare *cin* and *cout* as channels. All tasks that take *cin* by value, can read the input. The task that takes *cout* by reference, can write to the output. To do this, we allow the *main* function to take parameters. *main* takes *cin* by value and *cout* by reference. A "hello world" program in SHIM will look like this:

```
void main(chan char cin, chan char &cout) {
     cout << 'H';
     cout << 'e';
     cout << 'l';
     ..
     ..
     cout << 'l';
     cout << 'd';
}
```

The SHIM scheduler is a part of the runtime environment of the SHIM target program. It runs the asynchronous (communication-free) parts of the program independently — allowing the environment (operating system, hardware, etc.) to schedule these asynchronous sections of tasks with arbitrary interleavings. However, the SHIM scheduler will not violate the interthread communication rules forcing communication actions to synchronize.

All legal SHIM programs must be provably scheduling independent. For example, the *cell* function below implements a one-place buffer with an infinite loop that alternatively reads from its input channel and writes to its output channel. Then, by combining recursion and parallel composition, the *fifo* function chains *n* one-place buffers to build a fifo of size *n*.

```
void cell(chan int i, chan int &o) {
  while (true) { recv i; o = i; send o; }
}
void fifo(chan int i, chan int &o, int n) {
  chan int c; chan int m; m = n - 1;
  if (m>0) { cell(i, c); par fifo(c, o, m); }
  else { cell(i, o); }
}
```

The distribution of data tokens in the fifo is under the control of the scheduler. For instance, one scheduling policy may chose to move data tokens toward the output of the fifo eagerly; another may move data tokens lazily. Nevertheless, because this is a legal SHIM program, we know that the output of the fifo will always be the same for a particular input sequence.

SHIM also has an exception mechanism that is layered on top of its communication mechanism to preserve determinism.

```
void source(chan int &a) throws T {
  while (a > 0) {
    a = a - 1;
    send a;
  }
  throw T;
}
void sink(chan int b) {
  while (1)
        recv b;
}
void main() {
  chan int x = 5;
  try {
     source(x); par sink(x);
  } catch (T) {}
}
```

The *source* procedure in the above piece of code sends 4, 3, 2, 1, and 0 to the *sink*. The *sink* procedure calls *recv* five times to synchronize with the *source*'s sends. Then, *source* throws an exception *T*. When *sink* tries to receive the sixth time, it is poisoned by the *source* and terminated. It should be noted that the *sink* receives the poison only when it tries to communicate with *source*. As described, exceptions are propagated to other tasks only during communication, making the exception model of SHIM deterministic. We discuss a few more examples of SHIM programs with exceptions in the following chapters.

The central hypothesis of SHIM is that its simple, deterministic semantics helps both programming and automated program analysis. That we have been able to devise truly effective mechanisms for clever code generation and analysis (e.g., deadlock detection) that can gain deep insight into the behavior of programs, vindicates this view. The bottom line: if a programming language does not have simple semantics, it is really hard to analyze its programs quickly or precisely.

In the following chapters, we describe a series of code generation techniques suitable for parallel processors. Each actually works on a slightly different dialect of the SHIM language, although all use the Kahn-with-rendezvous communication scheme. The reason for this diversity is historical; we added features to the SHIM model as we discovered the need for them. Our benchmarks are all batch programs; we do not yet deal with reactive systems.

Chapter 4

Compiling SHIM to a Shared Memory Architecture

We have described the SHIM programming language in the previous chapter. To prove that the language can be practical, we describe a compiler that generates C code and calls the *Pthread* library for parallelism.

As discussed in the previous chapter, the SHIM language [47; 116] only allows deterministic message-passing communication to guarantee race freedom. The programming model allows SHIM compilers to use a simple syntactic check to verify that runtime scheduling choices cannot change a program's IO behavior. While this model does restrict how concurrent tasks may interact, the burden for the programmer and the performance penalty are a small price for correctness.

In this chapter, we demonstrate how SHIM facilitates writing interesting, time efficient parallel programs for shared-memory multiprocessors. The challenge is minimizing overhead - implementing SHIM's multiway rendezvous communication with exceptions efficiently is the main code generation challenge. Each communication action acquires the lock on a channel, checks whether every connected process had also blocked (whether the rendezvous could occur), and then checks if the channel is connected to a poisoned process (an exception had been thrown).

We implement a parallel JPEG decoder and an FFT to show how SHIM helps with coding and testing different schedules during design exploration (Section 4.2). We present a compiler that generates C code that calls the POSIX thread ("Pthread") library for shared-memory multiprocessors (Section 4.3). For the JPEG and FFT examples, our compiler's output achieves 3.05 and 3.3× speedups on a four-core processor (Section 4.4).

```
void h(chan int &A) {
  A = 4; send A;
  A = 2; send A;
}

void j(chan int A) throws Done {
  recv A;
  throw Done;
}

void f(chan int &A) throws Done {
  h(A); par j(A);
}
```

```
void g(chan int A) {
  recv A;
  recv A;
}

void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}
```

Figure 4.1: A concurrent SHIM program with communication and exceptions

4.1 Reviewing SHIM

SHIM [116] is a concurrent programming language designed to guarantee scheduling independence. The input-output function of a SHIM program does not depend on scheduling choices; that is, if two concurrent tasks are ready to run, choosing which to run first does not affect the program's function.

It adopts an asynchronous concurrency model, à la Kahn networks [70] (SHIM tasks can only block on a single channel), that uses CSP-like rendezvous [61]. The language does not expose shared memory to the programmer, but it does provide single-sender multiple-receiver synchronous communication channels and asynchronous exceptions. Both mechanisms were designed to prevent scheduling decisions from affecting function.

SHIM's syntax is a C subset augmented with constructs for concurrency, communication, and exceptions. It has functions with by-value and by-reference arguments, but no global variables, pointers, or recursive types.

The *par* construct starts concurrent tasks. *p par q* starts statements *p* and *q* in parallel, waits for both to complete, then runs the next statement in sequence.

To prevent data races, SHIM forbids a variable to be passed by reference to two concurrent tasks. For example,

```
void f(int &x) {}      void g(int x) {}

void main() {
  int x, y;
  f(x); par g(x); par f(y);                                        // OK
  f(x); par f(x);                 // rejected because x is passed by reference twice
}
```

Internally, our compiler only supports parallel function calls. If *p* in *p par q* is not a function call, *p* is transformed into a function whose interface—the formal arguments and whether they are by-reference or by-value—is inferred [116].

SHIM's channels enable concurrent tasks to synchronize and communicate without races. The *main* function in Figure 4.1 declares the integer channel *A* and passes it to *f* and *g*, then *f* passes it to *h* and *j*. Tasks *f* and *h* send data with *send A*. Tasks *g* and *j* receive it with *recv A*.

A channel resembles a local variable. Passing a channel by value copies its value, which can be modified independently. A channel must be passed by reference to senders.

Communication is blocking: a task that attempts to communicate must wait for all other connected tasks to engage in the communication. If the synchronization completes, the sender's value is broadcast to the receivers. In Figure 4.1, 4 is broadcast from *h* to *g* and *j*. Task *g* blocks on the second *send A* because task *j* does not run a matching *recv A*.

Like most formalisms with blocking communication, SHIM programs may deadlock. But deadlocks are easier to fix in SHIM because they are deterministic: on the same input, a SHIM program will either always or never deadlock.

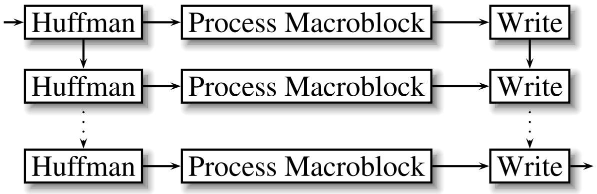


Figure 4.2: Dependencies in JPEG decoding

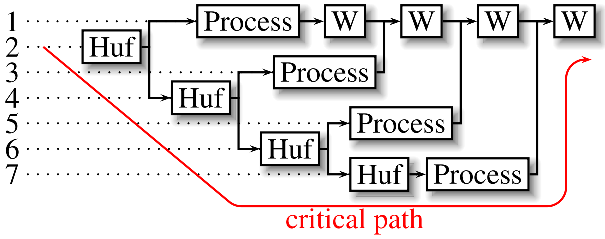


Figure 4.3: Seven-task schedule for JPEG

SHIM's exceptions enable a task to gracefully interrupt its concurrently running siblings. A sibling is "poisoned" by an exception when it attempts to communicate with a task that raised an exception or with a poisoned task. For example, when *j* in Figure 4.1 throws *Done*, it interrupts *h*'s blocked *send A* and *g*'s blocked *recv A*. An exception handler runs after all the tasks in its scope have terminated or been poisoned.

4.2 Design exploration with SHIM

SHIM facilitates the coding and testing of different schedules—a key aspect of design exploration for parallel systems. To illustrate, we describe implementing two parallel algorithms in SHIM: a JPEG decoder and an FFT.

4.2.1 Porting and parallelizing a JPEG decoder

We started by porting into SHIM a sequential JPEG decoder written in C by Pierre Guerrier. SHIM is not a C subset, so some issues arose. The C code held Huffman tables in global variables, which SHIM does not support, so we passed the tables explicitly. The C code allocated buffers with *malloc*; we used fixed-size arrays. We discarded a pointer-based Huffman decoder, preferring instead one that used arrays.

After some preprocessing, the main loop of the original program unpacked a macroblock—six Huffman-encoded 8×8 data blocks (standard 4:2:0 downsampling)—performed an IDCT on each data block, converted from YUV to RGB, and blitted the resulting 16×16 pixel block to a framebuffer. It then wrote the framebuffer to a file. Although macroblocks can be processed independently, unpacking and writing are sequential (Figure 4.2).

We first ran four IDCT transformers in parallel. Unfortunately, this ran slowly because of synchronization overhead.

To reduce overhead, our next version divided the image into four stripes and processed each independently. Fearing the cost of communication, we devised the seven-task schedule in Figure 4.3, which greatly reduced the number of synchronizations at the cost of buffer memory.

The Figure 4.3 schedule only gave a 1.8× speedup because the seventh task waits for all the other stripes to be unpacked and then everything waits for the seventh task. The arrow in Figure 4.3 shows the critical path, which includes the total cost of Huffman decoding and 14 of the IDCTs.

To strike a balance between the two approaches, we finally settled on the more fined-grained schedule in Figure 4.5. Each task processes a row of macroblocks at a time (e.g., 64 macroblocks for a 1024-pixel-wide image). This schedule spends less time waiting than the stripe-based approach and synchronizes less often than the block-based approach.

```
void unpack(unpacker_state &state, stripe &stripe) { ... }
void process(const stripe &stripe, pixels &pixels) { ... }
void write(writer_state &wstate, const pixels &pixels) { ... }

unpacker_state ustate; writer_state wstate;
stripe stripe1, stripe2, stripe3, stripe4;
pixels pixels1; chan pixels pixels2, pixels3, pixels4;

unpack(ustate, stripe1);
{ process(stripe1, pixels1); write(wstate, pixels1);
  recv pixels2; write(wstate, pixels2);
  recv pixels3; write(wstate, pixels3);
  recv pixels4; write(wstate, pixels4);
} par {
  unpack(ustate, stripe2);
  { process(stripe2, pixels2); send pixels2;
  } par {
    unpack(ustate, stripe3);
    { process(stripe3, pixels3); send pixels3;
    } par {
      unpack(ustate, stripe4);
      process(stripe4, pixels4); send pixels4;
} } }
```

Figure 4.4: SHIM code for the schedule in Figure 4.3

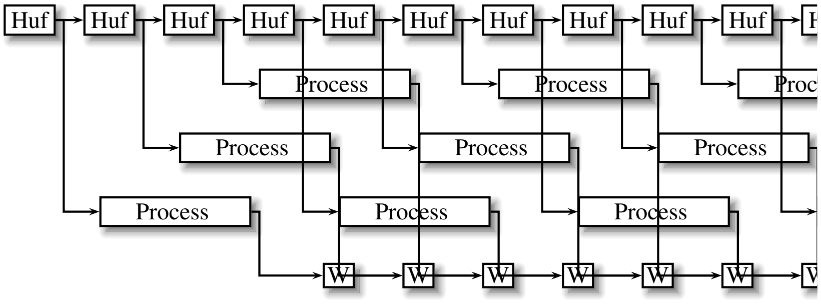


Figure 4.5: A pipelined schedule for JPEG

```
void unpack(unpacker_state &state, row &row) { ... }
void process(in row row, out pixels &pixels)
{ for (;;) { recv row; /* IDCT etc. */ send pixels; } }
void write(writer_state wstate, const pixels &pixels) { ... }

unpacker_state ustate; writer_state wstate; int rows;
chan row row1, row2, row3;
chan pixels pixels1, pixels2, pixels3;

try {
  { for (;;) {
      unpack(ustate, row1); send row1; if (--rows == 0) break;
      unpack(ustate, row2); send row2; if (--rows == 0) break;
      unpack(ustate, row3); send row3; if (--rows == 0) break;
    } throw Done; } par
  process(row1, pixels1); par
  process(row2, pixels2); par
  process(row3, pixels3); par
  { for (;;) {
      recv pixels1; write(wstate, pixels1);
      recv pixels2; write(wstate, pixels2);
      recv pixels3; write(wstate, pixels3); } }
} catch (Done) {}
```

Figure 4.6: SHIM code for the JPEG schedule in Figure 4.5

4.2.2 Parallelizing an FFT

We also coded in SHIM a pipelined FFT to test the effects of numerical roundoff. Its core is the FFT from *Numerical Recipes* [101], which we rewrote to use signed 4.28 fixed-point arithmetic. We added code that parses a .wav file, runs blocks of 1024 16-bit samples through the FFT, through an inverse FFT, then writes the samples to another .wav file.

Our FFT uses a schedule similar to that of the more complex JPEG decoder: one task reads 1024-sample blocks and feeds them to four FFT tasks in a round-robin manner. Each reads its sample block, performs the FFTinverse FFT operation, and sends its block to a writer task, which receives sample blocks in order and writes them sequentially.

Synchronization costs limited this to a 2.3× speedup on four processors, so we made it process 16 1024-sample blocks, improving performance to 3.3×.

4.2.3 Race freedom

Both the JPEG and FFT examples illustrate that dividing and scheduling computation tasks is critical in achieving performance on parallel hardware. Although data dependencies in JPEG were straightforward, finding the right schedule took some effort. With traditional concurrent formalisms, it is easy to introduce data races during design exploration.

SHIM's channels and exceptions cannot introduce races. E.g., in Figure 4.6, the first task throws an exception after reading all the rows. SHIM semantics ensure that the three row-processing tasks and the writing task terminate just after they have completed processing all the rows.

SHIM also guarantees data dependencies are respected. For instance, the SHIM compiler rejects attempts to run unpackers in parallel because of the shared pass-by-reference state (mostly, position in the file):

```
void unpack(unpacker_state &state, stripe &stripe) { ... }

unpack(ustate, stripe1); par unpack(ustate, stripe2);            // rejected
```

4.3 Generating Pthreads code for SHIM

In this section, we describe our main technical contribution: a SHIM compiler that generates parallel C code that uses the Pthread library's threads (independent program counters and stacks that share program and data memory), mutexes (mutual exclusion objects for synchronizing access to shared memory), and condition variables (can block and resume execution of other threads).

4.3.1 Mutexes and condition variables

Any Pthreads program must decide how many threads it will use, the number of mutexes, the partition of shared state, and the number and meaning of condition variables. These are partly engineering questions: coarse-grain locking leads to fewer locking operations but more potential for contention; finer locking has more overhead. Locking is fairly cheap, typically consisting of a (user-space) function call containing an atomic test-and-set instruction, but is not free. On one machine, locking and unlocking a mutex took 74× as long as a floating point multiply-accumulate.

We generate code that uses one mutex-condition variable pair for each task and for each channel. Figure 4.7 shows the data structures we use. These are "base classes:" the type of each task and channel includes additional fields that hold the formal arguments passed to the task and, for each function to which a channel is passed by value, a pointer to the local copy of the channel's value. To reduce locking, we track exception "poisoning" in both tasks and channels.

```
#define lock(m) pthread_mutex_lock(&m)
#define unlock(m) pthread_mutex_unlock(&m)
#define wait(c, m) pthread_cond_wait(&c, &m)
#define broadcast(c) pthread_cond_broadcast(&c)

enum state { STOP, RUN, POISON };
struct task {                          struct channel {
  pthread_t thread;                      pthread_mutex_t mutex;
  pthread_mutex_t mutex;                 pthread_cond_t cond;
  pthread_cond_t cond;                   unsigned int connected;
  enum state state;                      unsigned int blocked;
  unsigned int attached_children;        unsigned int poisoned;
  /* Formal arguments... */              /* Local copy pointers... */
};                                     };
```

Figure 4.7: Shared data structures for tasks and channels

4.3.2 The static approach

For efficiency, our compiler assumes the communication and call graph of the program is static. We reject programs with recursive calls, allowing us to transform the call graph into a call tree. This duplicates code to improve performance: fewer channel aspects are managed at run time.

We encode in a bit vector the subtree of functions connected to a channel. Since we know at compile time which functions can connect to each channel, we assign a unique bit to each function on a channel. We check these bits at run time with logical mask operations. In the code, something like *A_f* is a constant that holds the bit our compiler assigns to function *f* connected to channel *A*, such as 0x4.

4.3.3 Implementing rendezvous communication

Implementing SHIM's multiway rendezvous communication with exceptions is the main code generation challenge.

The code at a send or receive is straightforward: it locks the channel, marks the function and its ancestors as blocked, calls the *event* function for the channel to attempt the communication, and blocks until communication has occurred. If it was poisoned, it branches to a handler. Figure 4.8 is the code for *send A* in *h* in Figure 4.1.

```
lock(A.mutex);                        /* acquire lock for channel A */
A.blocked |= (A_h|A_f|A_main);        /* block h and ancestors on A */
event_A();                            /* alert channel of the change */
while (A.blocked & A_h) {                 /* while h remains blocked */
  if (A.poisoned & A_h) {                      /* were we poisoned? */
    unlock(A.mutex); goto _poisoned;
  }
  wait(A.cond, A.mutex);                       /* wait on channel A */
}
unlock(A.mutex);                         /* release lock for channel A */
```

Figure 4.8: C code for *send A* in function *h()*

For each channel, our compiler generates an *event* function that manages communication. Our code calls an *event* function when the state of a channel changes, such as when a task blocks or connects to a channel.

Figure 4.9 shows the *event* function our compiler generates for channel *A* in Figure 4.1. While complex, the common case is quick: when the channel is not ready (one connected task is not blocked on the channel) and no task is poisoned, *A.connected* != *A.blocked* and *A.poisoned* == 0 so the bodies of the two *if* statements are skipped.

If the channel is ready to communicate, *A.blocked* == *A.connected* so the body of the first *if* runs. This clears the channel (*blocked* = 0) and *main*'s value for *A* (passed by reference to *f* and *h* and passed by value to others) is copied to *g* or *j* if connected.

If at least one task connected to the channel has been poisoned, *A.poisoned* != 0 so the body of the second *if* runs. This code comes from unrolling a recursive procedure at compile time, which is possible because we know the structure of the channel (i.e., which tasks connect to it). The speed of such code is a key advantage over a library.

```
void event_A() {
  unsigned int can_die = 0, kill = 0;
  if (A.connected == A.blocked) {                        /* communicate */
    A.blocked = 0;
    if (A.connected & A_g) *A.g = *A.main;
    if (A.connected & A_j) *A.j = *A.main;
    broadcast(A.cond);
  } else if (A.poisoned) {                        /* propagate exceptions */
    can_die = blocked & (A_g|A_h|A_j);              /* compute can_die set */
    if (can_die & (A_h|A_j) == A.connected & (A_h|A_j))
       can_die |= blocked & A_f;
    if (A.poisoned & (A_f|A_g)) {                        /* compute kill set */
      kill |= A_g; if (can_die & A_f) kill |= (A_f|A_h|A_j);
    }
    if (A.poisoned & (A_h|A_j)) { kill |= A_h; kill |= A_j; }
    if (kill &= can_die & ~A.poisoned) {            /* poison some tasks? */
      unlock(A.mutex);
      if (kill & A_g) {                            /* poison g if in kill set */
        lock(g.mutex);
          g.state = POISON;
        unlock(g.mutex); }
      /* also poison f, h, and j if in kill set... */
      lock(A.mutex);
        A.poisoned |= kill; broadcast(A.cond);
} } }
```

Figure 4.9: C code for the *event* function for channel *A*

```
lock(main.mutex); main.state = POISON; unlock(main.mutex);
lock(f.mutex); f.state = POISON; unlock(f.mutex);
lock(j.mutex); j.state = POISON; unlock(j.mutex);
goto _poisoned;
```

Figure 4.10: C code for *throw Done* in function *j()*

This exception-propagation code attempts to determine which tasks, if any, connected to the channel should be poisoned. It does this by manipulating two bit vectors. A task *can_die* iff it is blocked on the channel and all its children connected to the channel (if any) also *can_die*. A *poisoned* task may *kill* its sibling tasks and their descendants. Finally, the code kills each task in the *kill* set that *can_die* and was not *poisoned* before by setting its *state* to *POISON* and updating the channel accordingly (*A.poisoned* |= *kill*).

Code for throwing an exception (Figure 4.10) marks as *POISON* all its ancestors up to where it will be handled. Because the compiler knows the call tree, it knows how far to "unroll the stack," i.e., how many ancestors to poison.

4.3.4 Starting and terminating tasks

It is costly to create and destroy a POSIX thread because it usually requires a system call, each has a separate stack, and doing so interacts with the operating system's scheduler. To minimize this overhead, because we know the call graph of the program at compile time, our compiler generates code that creates at the beginning as many threads as the SHIM program will ever need. These threads are only destroyed when the SHIM program terminates; if a SHIM task terminates, its POSIX thread blocks until it is reawakened.

```
lock(A.mutex); /* connect */
  A.connected |= (A_f|A_g);
  event_A();
unlock(A.mutex);

lock(main.mutex);
  main.attached_children = 2;
unlock(main.mutex);

lock(f.mutex); /* pass args */
  f.A = &A;
unlock(f.mutex);

/* A is dead on entry for g,
   so do not pass A to g */
```

```
lock(f.mutex);          /* run f() */
  f.state = RUN; broadcast(f.cond);
unlock(f.mutex);

lock(g.mutex);          /* run g() */
  g.state = RUN; broadcast(g.cond);
unlock(g.mutex);

lock(main.mutex); /* wait for children */
  while (main.attached_children)
    wait(main.cond, main.mutex);
  if (main.state == POISON) {
    unlock(main.mutex);
    goto _poisoned; }
unlock(main.mutex);
```

Figure 4.11: C code for calling *f()* and *g()* in *main()*

Figure 4.11 shows the code in *main* that runs *f* and *g* in parallel. It connects *f* and *g* to channel *A*, sets its number of live children to 2, passes function parameters, then starts *f* and *g*. The address for the pass-by-reference argument *A* is passed to *f*. Normally, a value for *A* would be passed to *g*, but our compiler found this value is not used so the copy is avoided (discussed below). After starting *f* and *g*, *main* waits for both children to return. Then *main* checks whether it was poisoned, and if so, branches to a handler.

```
 int *A; /* value of channel A */

_restart:
 lock(f.mutex);
   while (f.state != RUN)
     wait(f.cond, f.mutex);
   A = f.A; /* copy arg. */
 unlock(f.mutex);

 /* body of the f task */

_terminated:
 lock(A.mutex); /* disconnect f */
   A.connected &= ~A_f;
   event_A();
 unlock(A.mutex);

 lock(f.mutex); /* stop */
   f.state = STOP;
 unlock(f.mutex);
 goto _detach;
```

```
_poisoned:
 lock(A.mutex); /* poison A */
   A.poisoned |= A_f;
   A.blocked &= ~A_f; event_A();
 unlock(A.mutex);

 lock(f.mutex); /* wait for children */
   while (f.attached_children)
     wait(f.cond, f.mutex);
 unlock(f.mutex);

 lock(A.mutex); /* disconnect j, h */
   A.connected &= ~(A_h|A_j);
   A.poisoned &= ~(A_h|A_j);
   event_A();
 unlock(A.mutex);

_detach: /* detach from parent */
 lock(main.mutex);
   --main.attached_children;
   broadcast(main.cond);
 unlock(main.mutex);
 goto _restart;
```

Figure 4.12: C code in function *f()* controlling its execution

Reciprocally, Figure 4.12 shows the code in *f* that controls its execution: an infinite loop that waits for *main*, its parent, to set its *state* field to running, at which point it copies its formal arguments into local variables and runs its body.

If a task terminates normally, it cleans up after itself. In Figure 4.12, task *f* disconnects from channel A, sets its *state* to *STOP*, and informs *main* it has one less running child.

By contrast, if a task is poisoned, it may still have children running and it may also have to poison sibling tasks so it cannot entirely disappear yet. In Figure 4.12, task *f*, if poisoned, does not disconnect from *A* but updates its *poisoned* field. Then, task *f* waits for its children to return. At this time, *f* can disconnect its (potentially poisoned) children from channels, since they can no longer poison siblings. Finally, *f* informs *main* it has one less running child.

4.3.5 Optimizations

SHIM draws no distinction between sequential C-like functions and concurrent tasks; our compiler treats them differently for efficiency. Our compiler distinguishes tasks from functions, which must not take any channel arguments, contain local channels, throw or handle exceptions, have parallel calls, call any tasks, or be called in parallel. Tasks are implemented as described above—each is assigned its own thread. Functions follow C's calling conventions.

Unlike Java, SHIM passes scalars, structures, and arrays by value unless marked as by-reference. This is convenient at parallel call sites to avoid interference among concurrent tasks. However, if tasks only read some data, the data can be shared among them for efficiency. Similarly, a channel can be shared among tasks that never update the channel's value between *recv* instructions. We introduced a C++-like *const* specifier that prohibits assignments to a variable, channel, or function parameter. The compiler allows multiple concurrent *const* by-reference parameters and allocates a shared copy for *const* parameters passed by value.

We implemented another optimization to reduce superfluous copies of large data structures. Normally, the current value of a channel is copied when the channel is passed by value, but copying is unnecessary if the value is never used before the next value is *recv*'d. The overhead can be substantial for arrays. We perform live variable analysis to determine which arguments are dead on entry. E.g., in

```
void myfunc(chan int input[65536]) { recv input; ... }
```

the *input* channel value is dead on entry and will not be copied at any callsite for *myfunc*, eliminating a 256K copy.

4.4 Experimental results

We implemented our SHIM compiler in OCAML. Code specific to the Pthreads backend is only about 2000 lines.

To test the performance of our generated code, we ran it on a 1.6 GHz Quad-Core Intel Xeon (E5310) server running Linux kernel 2.6.20 with SMP (Fedora Core 6). The processor "chip" actually consists of two dice, each containing a pair of processor cores. Each core has a 32 KB L1 instruction and a 32 KB L1 data cache, and each die has a 4 MB of shared L2 cache shared between the two cores.

We compiled the generated C with gcc 4.1.1 with *-O7* and *-pthread* options. We timed it using the *time* command and ran *sync* to flush the disk cache.

The JPEG program uses much more stack space than typical C programs because it stores all data on the stack instead of the heap. We raised the stack size to 16 MB with *ulimit -s*.

Table 4.1: Experimental Results for the JPEG decoder

Cores	Tasks	Time	Total	TotalTime	Speedup
1	†	25s	20s	0.8	1.0× (def)
1	1+3+1	24	24	1.0	1.04
2	1+3+1	13	24	1.8	1.9
3	1+3+1	11	24	2.2	2.3
4	1+3+1	8.7	25	2.9	2.9
4	1+1+1	16	24	1.5	1.6
4	1+2+1	9.3	25	2.7	2.7
4	1+3+1	8.7	25	2.9	2.9
4	1+4+1	8.2	25	3.05	3.05
4	1+5+1	8.6	25	2.9	2.9

† Reference single-threaded C implementation.
Run on a 20 MB 21600 × 10800 image that expands to 668 MB. Tasks is the number of parallel threads (read and unpack + process row + write), Time is wallclock, Total is user + system time, TotalTime is the parallelization factor, speedup is with respect to the reference implementation.

Table 4.1 shows results for the JPEG decoder. We ran it on a 20 MB earth image from NASA[1] and varied both the number of available processors and the number of row-processing tasks in our program. The speedup due to parallelization plateaued at 3.05, which we attribute to the sequential nature of the Huffman decoding process.

Table 4.2 shows statistics for our FFT. We compared handwritten C with sequential SHIM and two parallel SHIM versions, one with six tasks that work on single 1024-sample blocks and one that works on sixteen such blocks. The first parallel implementation has overhead from synchronization and communication. The "Parallel 16" version communicates less to reduce this overhead and achieve a 3.3× speedup: 82% of an ideal 4× speedup on four cores.

4.5 Related work

Like SHIM, the StreamIt language [120] is deterministic, but its dataflow model is a strict subset of SHIM's and there is no StreamIt compiler for shared memory machines.

Other concurrent languages use different models. The most common is "loops-over-arrays," embodied, e.g., in compilers for OpenMP [121]. This would be

[1] world.200409.3x21600x10800.jpg from earthobservatory.nasa.gov

Table 4.2: Experimental Results for the FFT

Code	Cores	Time	Total	TotalTime	Speedup
Handwritten C	1	2.0s	2.0s	1.0	1.0× (def)
Sequential SHIM	1	2.1	2.1	1.0	0.95
Parallel SHIM	1	2.1	2.1	1.0	0.95
Parallel SHIM	2	1.3	2.0	1.5	1.5
Parallel SHIM	3	0.92	2.1	2.2	2.2
Parallel SHIM	4	0.86	2.1	2.4	2.3
Parallel 16	1	1.9	1.9	1.0	1.1
Parallel 16	2	1.0	1.9	1.9	2.0
Parallel 16	3	0.88	1.9	2.1	2.2
Parallel 16	4	0.6	1.9	3.2	3.3

Run on a 40 MB audio file—20 000 1024-point FFTs.

awkward for a schedule such as Figure 4.5. The Cilk language [19] speculates to parallelize sequential code. The Guava [9] Java dialect prevents unsynchronized access to shared objects by enforcing monitor use with a type system. Like SHIM, it aims for race freedom, but uses a very different model.

4.6 Conclusions

A good parallel algorithm reliably computes the result quickly. Unlike most parallel languages, SHIM guarantees reliability by preventing data races. Correctness remains a challenge, but at least running a SHIM program on a test case gives consistent results for any scheduling policy.

SHIM is helpful during design exploration when testing different schedules; its determinacy makes it easy to obey data dependencies. Its C-like syntax facilitates porting existing code. We demonstrated this on a JPEG decoder.

Our SHIM compiler generated code for parallel programs that runs on a four-core processor over three times faster than sequential C. Sequential SHIM code runs no slower. We therefore believe that SHIM can be practical. We strengthen this argument by generating code for a heterogeneous architecture in the next chapter.

Chapter 5

Compiling SHIM to a Heterogeneous Architecture

In the previous chapter, we demonstrated that the SHIM model can be practical for a shared memory architecture. In this chapter, we evaluate the model for a different parallel architecture: the Cell Broadband Engine.

The Cell architecture is interesting but is notoriously difficult to program. In addition to the low-level constructs (e.g., locks, DMA), it allows most parallel programming environments to admit data races: the environment may make nondeterministic scheduling choices that can change the function of a program.

In this chapter, we describe a compiler for the SHIM scheduling-independent concurrent language that generates code for the Cell Broadband heterogeneous multicore processor. The complexity of the code our compiler generates relative to the source illustrates how difficult it is to manually write code for the Cell.

Our backend [124] is a direct offshoot of the pthreads backend but allows the user to assign certain (computationally intensive) tasks directly to the CELL's eight synergistic processing units (SPEs); the rest of the tasks run on the CELL's standard PowerPC core (PPE). Our technique replaces the offloaded functions with wrappers that communicate across the PPE-SPE boundary. Cross-boundary function calls are technically challenging because of data alignment restrictions on function arguments, which we would have preferred to be stack resident. This, and many other fussy aspects of coding for the CELL, convinced us that such heterogeneous multicore processors demand languages at a higher level than sequential software.

We demonstrate the efficacy of our compiler on two examples. While the SHIM language is (by design) not ideal for every algorithm, it works well for certain applications and simplifies the parallel programming process, especially on the Cell architecture.

We review the Cell processor, and describe the inner workings of our compiler. In Section 5.3, we describe how we instrumented our generated code to collect performance data, and present experimental results in Section 5.4.

5.1 The Cell Processor

Coherent shared memory multiprocessors, such as the Intel Core Duo, follow a conservative evolutionary path. Unfortunately, maintaining coherence costs time, energy, and silicon because the system must determine when data is being shared, and relaxed memory ordering models [1] make reasoning about coherence difficult.

The Cell processor [98; 69; 72], the target of our compiler, instead uses a heterogeneous architecture consisting of a traditional 64-bit power processor element (PPE) with its own 32K L1 and 512K L2 caches coupled to eight synergistic processor elements (SPEs).

Each SPE is an 128-bit processor whose ALU can perform up to 16 byte operations in parallel. Each has 128 128-bit general-purpose (vector) registers, a 256K local store, but no cache. Each SPE provides high, predictable performance on vector operations.

Our compiler uses multiple cores to provide task-level parallelism. Most cell compilers address the Cell's vector-style data parallelism [53].

Cell programs use direct-memory access (DMA) operations to transfer data among the PPE and SPEs' memories. While addresses are global (i.e., addresses for the PPE's and each SPE's memories are distinct), this is not a shared memory model. That our compiler relieves the programmer from having to program the Cell's memory flow controllers (DMA units) is a key benefit.

5.1.1 DMA and Alignment

The centerpiece of the Cell's communication system—and a major concern of our compiler—is the element interconnect bus (EIB): two pairs of counter-rotating rings [72; 4], each 128 bits (16 bytes—a quadword) wide.

The width of the EIB leads the DMA units to operate on 128-bit-wide memory. Memory remains byte-addressed, but the 128-bit model puts substantial constraints on transfers because of the lack of byte-shifting circuitry [64, p. 61].

A DMA unit most naturally transfers quadwords. It can copy between 1 and 1024 quadwords (16K) per operation; source and destination addresses must be quadword aligned.

A DMA unit can also transfer 1, 2, 4, or 8 bytes. The source and destination addresses must be aligned on the transfer width and have the same alignment within quadwords. For example, a 7-byte transfer requires three DMA operations, and transferring a byte from address 3 to address 5 requires a DMA to a buffer followed by a memory-to-memory move. To perform DMA operations, our compiler generates code that calls complex C macros that usually distill down to only a few machine instructions.

Our compiler produces C code suitable for the port of GCC to the SPE. We take advantage of a GCC extension that can place additional alignment constraints on types and variables. For example, a *struct* type or array variable can be constrained to start on a 16-byte boundary (e.g., to make it work with the DMA facility):

```
struct foo { int x, y; } __attribute__ ((aligned (16)));
int z[10] __attribute__ ((aligned (16)));
```

5.1.2 Mailboxes and Synchronization

For synchronization, our compiler generates code that uses the Cell's mailboxes: 32-bit FIFO queues for communication between the PPE and an SPE. Each SPE has two one-entry mailboxes for sending messages to the PPE and one four-entry queue for messages from the PPE [64, p. 101].

We use mailboxes for synchronization messages between the main program running on the PPE and tasks running on the SPEs. The SPE writing to an outbound mailbox causes an interrupt on the PPE, prompting it to read and empty the mailbox. In the other direction, the PPE writes to the SPE's inbound mailbox and can signal an interrupt on the SPE, but we just do a blocking read on the inbound SPE mailbox to wait for the next message.

All our communication is done using handshaking through the mailboxes; our protocol ensures the mailboxes do not overflow.

The Cell also provides signals: 32-bit registers whose bits can be set and read for synchronization; our code does not use them.

5.2 Our Compiler

We generate asymmetric code because of asymmetries in the Cell architecture and runtime environment. For example, the PPE supports pthreads but we do not know of a similar library for the SPEs. Also, mailboxes, the more flexible of the Cell's two synchronization mechanisms, work best between the PPE and an SPE. They can be used between SPEs, but are more awkward.

These considerations, along with our experience in implementing SHIM on shared-memory systems [51], led us to adopt a "computational acceleration" model [69] in which the SPEs run more time-critical processes and the PPE is responsible for the rest, including coordination among the SPEs. Communication in the code we generate takes place between the PPE and an SPE.

Figure 5.1 shows the structure of the code we generate, here for the small example from Figure 4.1 in the previous chapter. In Figure 4.1, the value 4 is broadcast from *h* to *g* and *j*. Task *g* blocks on the second ***send*** *A* because task *j* does not run a matching ***recv*** *A*.

We instructed our compiler to assign tasks *h* and *j* to two SPEs; all the others run on the PPE.

For PPE-resident tasks, our compiler generates almost the same pthreads-based code we presented in the previous chapter. For each SPE-resident task, we generate SPE-specific code that communicates through mailboxes and DMA to a proxy function running on the PPE (e.g., *_func_j* in Figure 5.1). The SPE functions, shown at the bottom of Figure 5.1, translate communication from the SPE code to the PPE-resident pthreads environment.

5.2.1 Code for the PPE

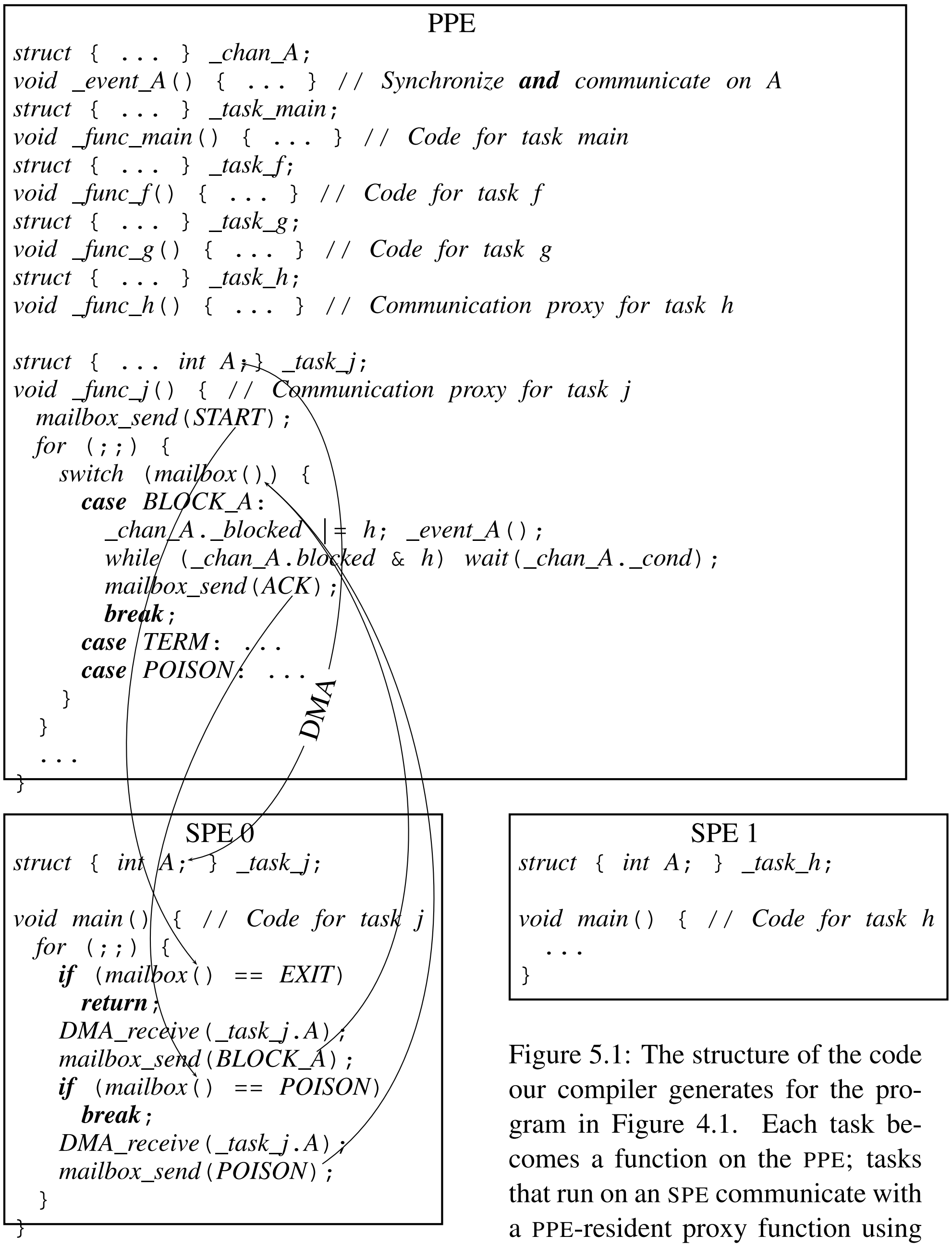
The C code we generate for the PPE uses the pthreads library to emulate concurrency much like we did for our shared-memory compiler [51]. Each task and each channel has its own shared data structure that includes a lock used to guarantee access to it is atomic and a condition variable for notifying other threads of state changes (Figure 5.2). Each of these channels resides in main (PPE) memory and are manipulated mostly by the PPE code.

For each SHIM function, our compiler generates a C function that runs in its own thread. For each channel, we generate an *event* function responsible for managing synchronization and communication on the channel (e.g., *_event_A* at the top of Figure 5.1). For speed, our compiler "hardwires" the logic of each *event* function because a SHIM program's structure is known at compile time. A generic function controlled by channel-specific data would be more compact but slower.

5.2.2 Code for the SPEs

For each SHIM function that will execute on an SPE, we generate a C function and compile it with the standard port of GCC to the SPEs. Again, most of SHIM is translated directly into C; code for communication and synchronization is the challenge.

Our strategy is to place most of the control burden on the PPE and use the SPEs to offload performance-critical tasks. This simplifies code generation by removing the need for inter-SPE synchronization; we only need an SPE-PPE mechanism.



```
struct { ... } _chan_A;
void _event_A() { ... } // Synchronize and communicate on A
struct { ... } _task_main;
void _func_main() { ... } // Code for task main
struct { ... } _task_f;
void _func_f() { ... } // Code for task f
struct { ... } _task_g;
void _func_g() { ... } // Code for task g
struct { ... } _task_h;
void _func_h() { ... } // Communication proxy for task h

struct { ... int A; } _task_j;
void _func_j() { // Communication proxy for task j
  mailbox_send(START);
  for (;;) {
    switch (mailbox()) {
      case BLOCK_A:
        _chan_A._blocked |= h; _event_A();
        while (_chan_A.blocked & h) wait(_chan_A._cond);
        mailbox_send(ACK);
        break;
      case TERM: ...
      case POISON: ...
    }
  }
  ...
}
```

```
struct { int A; } _task_j;

void main() { // Code for task j
  for (;;) {
    if (mailbox() == EXIT)
      return;
    DMA_receive(_task_j.A);
    mailbox_send(BLOCK_A);
    if (mailbox() == POISON)
      break;
    DMA_receive(_task_j.A);
    mailbox_send(POISON);
  }
}
```

```
struct { int A; } _task_h;

void main() { // Code for task h
  ...
}
```

Figure 5.1: The structure of the code our compiler generates for the program in Figure 4.1. Each task becomes a function on the PPE; tasks that run on an SPE communicate with a PPE-resident proxy function using mailboxes and DMA.

```
int foo(int a, int &b, chan uint8 cin, chan uint8 &cout) {
  next cout = a; next cout = b; next cout = next cin;
  return '\n';
}
struct {
  pthread_t _thread;
  pthread_mutex_t _mutex;
  pthread_cond_t _cond;
  enum _state { _STOPPED, _RUNNING, _POISONED } _state;
  unsigned int _attached_children;
  unsigned int _dying_children;
  int *b;
  int *__return_var;
  struct {
    struct {
      unsigned char cout;
      int b;
      int __return_var;
    } _byref;
    unsigned char cin;
    int a;
  } _args __attribute__((aligned (16)));
} _foo;

struct {
  pthread_mutex_t _mutex;
  pthread_cond_t _cond;
  unsigned int _connected;
  unsigned int _blocked;
  unsigned int _poisoned;
  unsigned int _dying;
  unsigned char *foo;
  unsigned char *main_1;
  unsigned char *main;
} _cin;
```

Figure 5.2: Shared data for the *foo* task and *cin* channel.

Using command-line arguments, the user specifies one or more "leaf" functions to run on the SPEs, such as tasks *h* and *j* in Figure 5.1. Such functions may communicate on channels, but may not start other functions in parallel or call functions that communicate. However, a leaf function may call other functions that do not communicate or invoke functions in parallel, i.e., those that behave like standard C functions. This restriction saves us from creating a mechanism for starting tasks from an SPE.

The pthreads synchronization mechanisms (mutexes, condition variables) our code uses do not work across the PPE/SPE boundary.[1] Instead, for each function destined for an SPE, we synthesize a proxy function on the PPE that acts as a proxy for the function on the SPE that does the actual work (*_func_j* and *_func_h* in Figure 5.1). Each proxy translates between pthreads events on the PPE and mailbox events from the SPE.

Passing arguments to an SPE task turns out to be awkward because of DMA-imposed alignment constraints. Our solution requires two copies: a DMA transfer from the PPE followed by word-by-word copying into local variables, which allows the compiler to optimize their access. This is one of the few cases where compiling into C is a disadvantage over generating assembly.

Channel communication is done through mailbox messages for synchronization and DMA for data transfer (Figure 5.1). It starts when the SPE task sends a BLOCK message to the PPE for a particular channel. This prompts the PPE proxy to signal it is blocked on that channel. When the *event* function on the PPE releases the channel (i.e., when all connected tasks have rendezvoused), the PPE sends an ACK message to the SPE, which prompts it to start a DMA transfer to copy the data for the channel from the argument *struct* on the PPE to a matching *struct* on the SPE. There is no danger of this data being overwritten because only the *event* function on the PPE writes into the *struct*, and that will only happen after the task is again blocked on the channel, which will not happen until the SPE task requests it, which will only happen after the DMA is complete.

A task may become "poisoned" when it attempts a rendezvous and another task in the same scope has thrown an exception. The *event* function in the PPE code handles the logic for propagating exception poison; the PPE proxy code is responsible for informing the SPE task it has been poisoned.

The SPE code may send two other messages. TERM is the simpler: the SPE sends this when it has terminated, and the PPE proxy jumps to its own *_terminate* handler, which informs its parent that it has terminated. The other message is POISON, which the SPE code sends when it throws an exception. After this, it

[1] IBM's "Example" library [65] does provide cross-processor mutexes, but blocking operations never yield to the thread scheduler.

sends another word that indicates the specific exception. Based on this word, the proxy marks itself and all its callers in the scope of the exception as poisoned, then jumps to the *_poisoned* label, which also handles the case where the task has been poisoned by a channel.

5.3 Collecting Performance Data

While tuning our compiler and applications, we found we needed pictures of the temporal behavior of our programs. While speeding up any part of a sequential program is beneficial, improving a parallel program's performance requires speeding computation along a critical path—any other improvement is hidden.

To collect the data we wanted, we added a facility to our compiler that collects the times at which communication events begin and end. For this, we use the SPE's "decrementer"—a high-speed (about 80 MHz) 32-bit software-controlled countdown timer. Our compiler can add code that reads this timer and stores the starting and stopping times of each communication action, i.e., periods when the SPE is blocked waiting for synchronization. We fill a small buffer in the SPE's local store, then dump the event timestamps into a text file when the program terminates. Our goal is to be as unintrusive; each sample event consists of testing whether the buffer is full, reading the timer, writing into an array, and incrementing a destination pointer.

To understand the interaction among SPEs, we wanted global time stamps, so we include code to synchronize the decrementers. Although the SPEs' decrementers run off a common clock, their absolute values are set by software and not generally synchronized.

Our synchronization code measures round-trip communication time and uses it to synchronize the clocks on the SPEs. We assign one SPE to be the master, then synchronize all the other SPEs' clocks to it. The master first establishes communication with the slave (i.e., waits for the slave to start), then sends a message to the slave through its mailbox, which immediately sends it back. The master measures the time this took—the round-trip time. Finally, the master sends the current value of its clock plus half the round-trip time to the slave, which sets its clock to that value.

Figures 5.3 and 5.4 shows data we obtained with this mechanism. Time runs from left to right, and each line segment denotes the time that one SPE is either blocked or communicating; empty spaces between horizontal lines indicate time an SPE is doing useful work. The vertical position of each line indicates the SPE number.

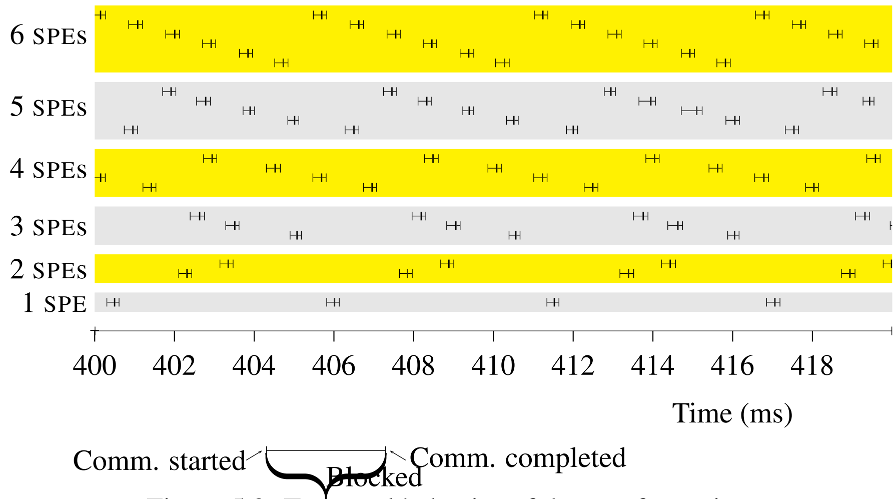


Figure 5.3: Temporal behavior of the FFT for various SPEs

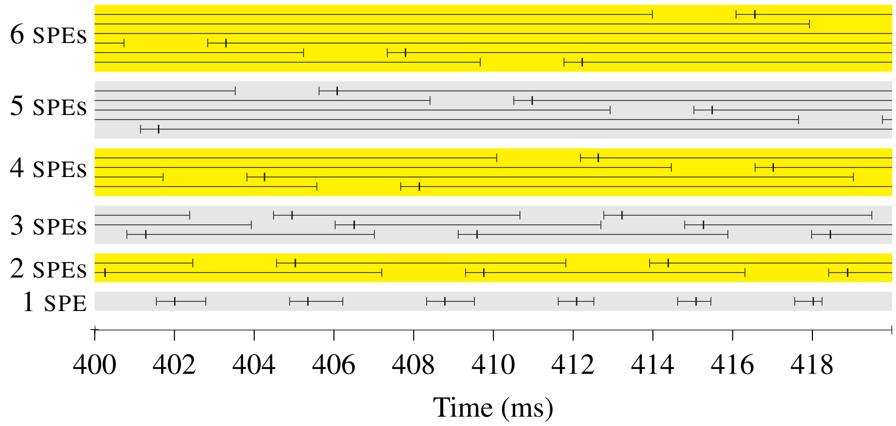


Figure 5.4: Temporal behavior of the JPEG decoder

5.4 Experimental Results

To evaluate our compiler, we used it to compile a pair of applications and ran them on a Sony Playstation 3 running Fedora Core 7 with Linux kernel 2.6.23 and the IBM SDK version 3.0.

The Sony Playstation 3 is a Cell-based machine with 256 MB of memory, a single Cell with one SPE disabled to improve yield, and peripherals including an Ethernet interface and a hard drive. While the PS3 platform is open enough to boot an operating system such as Linux, it does not allow full access to the hardware. Instead, guest operating systems run under a hypervisor that limits access to the hardware such as the disk, only part of which is visible to Linux. The hypervisor on the PS3 also reserves one of the SPEs for security tasks, leaving six available to our programs.

We compiled the generated C code with GCC 4.1.2 for the PPE and 4.1.1 for the SPE code, both optimized with -O.

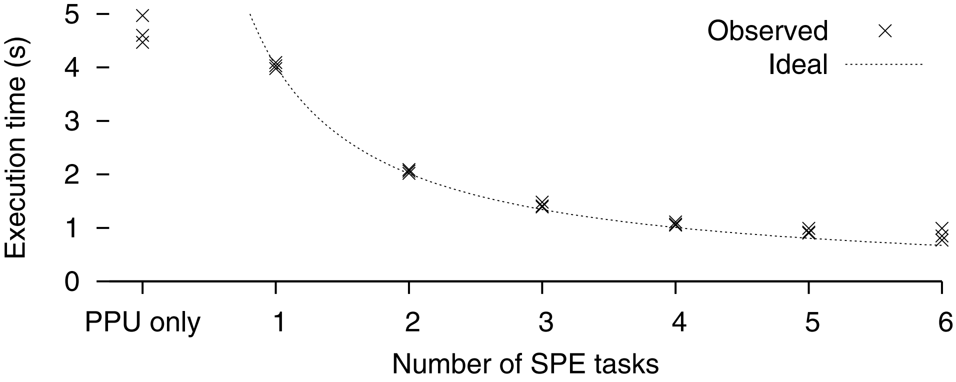


Figure 5.5: Running time for the FFT on varying SPEs
(Run on a 20 MB audio file, 1024-point FFTs)

Figure 5.5 shows execution times for an FFT that takes an audio file, divides it into 1024-sample blocks, performs a fixed-point (4.28) FFT on each block, follows it by an inverse FFT, and writes it out to a file. A PPE-based reader tasks distributes 8 1024-sample blocks to the SPE tasks in a round-robin order; a writer task collects them in order and writes them out to a file. We communicate 8 blocks instead of the 16 we used earlier [51] to accommodate the SPEs' local store. We ran this on a 20 MB stereo audio file with 16-bit samples. The "PPE only" code is from our earlier compiler [51].

Figure 5.3 illustrates why we observe a near-ideal speedup for the FFT on six SPEs. Roughly half the time all six are doing useful work; otherwise one is blocked communicating, giving a speed-up of about $11/2 = 5.5$, close to the 5.3 we observed (Figure 5.5).

Each horizontal line in Figure 5.3 represents two events: an FFT task on an SPE reads a block, processes it, sends it, and then repeats the process; the read immediately follow the write. The figure also shows that the processes spend more time blocking waiting to write than they do to read, suggesting the task that reassembles data from the FFT tasks is slower than the one that parcels it out.

We also compiled and ran a JPEG decoder, similar to our earlier work [51]. Figure 5.6 shows the execution times we observed, which do not exhibit the same speedup as the FFT and are much more varied. Figure 5.4 explains why: for these runs, the SPEs are spending most of their time waiting for data. For this sample, only at one point the 3-SPE case is more than one SPE active at any time.

Figure 5.4 tells us the SPEs are usually waiting for data to arrive. Each line segment is actually two parts: sending processed data (left), and receiving unprocessed data. This is not surprising; while JPEG data is composed of independent blocks, the data itself is Huffman encoded, meaning it requires the data to be uncompressed before block boundaries can be identified.

The performance figures we report are for carefully chosen problem sizes. Start-up overhead is larger for smaller problems sizes, leading to poorer results; the data for larger problem sizes does not fit into the PS3's 256 MB of main memory, necessitating disk access that quickly becomes the bottleneck. For large data sets, our performance degrades to just disk I/O bandwidth, suggesting the PS3 is not ideally suited to large scientific computing tasks.

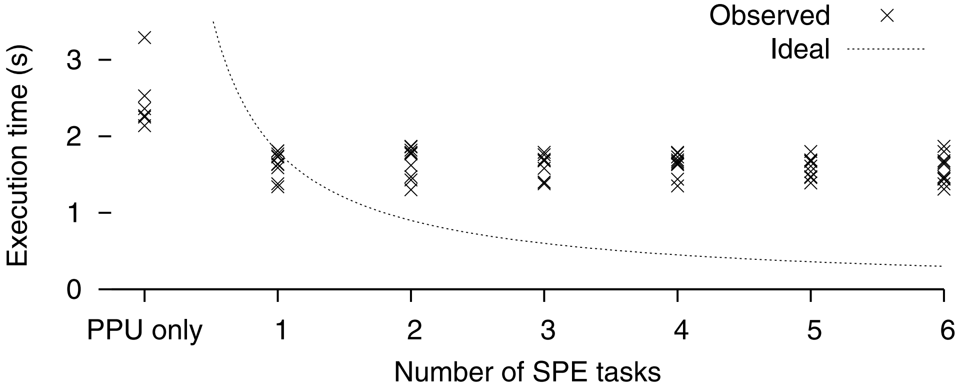


Figure 5.6: Running time for the JPEG decoder on varying SPEs
(Run on a 1.7 MB image that expands to a 29 MB raster file)

5.5 Related Work

Other groups that have produced compilers for the Cell start from models very different from SHIM and address different problems.

Eichenberger et al.'s compiler [53; 54] takes a traditional approach by starting with C code with OpenMP annotations [95] and generates code for the Cell. They consider low-level aspects of code generation: vectorizing scalar, array-based code; hiding branch latency; and ensuring needed data alignment. They implement the OpenMP model: programmers provide hints about parallelizable loops, then the compiler breaks these into separate tasks and distributes them to the SPEs. It presents a shared memory model, which their runtime system emulates with explicit DMA transfers.

OpenMP is a much different programming model than SHIM: it assumes shared memory and focuses on parallelizing loops with array access. SHIM, by contrast, is a stream-based language with explicit communication. Adding OpenMP-like constructs to improve SHIM's array performance would be a nice complement.

Adopting a more SHIM-like message passing approach, Ohara et al.'s [91] preprocessor takes C programs written using the standard message passing interface (MPI) API [85], determines a static task graph, clusters and schedules this graph, and finally regenerates the program to use Cell-specific API calls for communication.

Semantically, the MPI model is similar to SHIM but does not guarantee scheduling independence. The big difference is that the preprocessor of Ohara et al. does not enforce the programming style; it would be easy to write a misbehaving program. The SHIM compiler catches a host of bugs including deadlock [122].

Fatahalian et al.'s Sequoia [55] is most closely related to our work. Like us, they compile a high-level concurrent language to the Cell processor (and other architectures) with the goal of simplifying the development process.

Their underlying computational model differs substantially from SHIM's, however. While also explicitly parallel, it is based on stateless procedures that only receive data when they start and only transmit it when they terminate. This model, similar to the one in Cilk [19], is designed for divide-and-conquer algorithms that partition large datasets (typically arrays) into pieces, work on each piece independently, then merge the results. While our example applications also behave this way, other SHIM programs do not.

While the low-level compilation challenges of the Cell are fairly conventional, higher-level issues are less obvious. Because the processor is young and idiosyncratic, there is still work to be done in choosing strategies for structuring large programs. For example, Petrini et al. [97] observe a high performance implementation of a three-dimensional neutron transport algorithm requires a careful balance among vector parallelism in the SPEs, the effect of their pipelines, balancing and scheduling DMA operations, and coordinating multiple SPEs. Saidani et al. [105] change DMA transfer sizes to improve the performance of an image processing algorithm. Gedik et al. [56] optimize distributed sorting algorithms on the Cell by careful vectorization and communication. They note main memory bandwidth becomes the bottleneck on large datasets since the inter-SPE bandwidth is so high. Our compiler only provides higher-level data communication and synchronization facilities.

Chow et al. [32] discuss coding a large FFT on the Cell. They suggest putting the control of the application on the PPE, then offloading computationally intensive code to the SPEs and adapting it to work with the SPEs' vector capabilities. We adopt a similar philosophy in the code generated by our compiler.

They target their application at a 128 MB dataset—too large to fit in on-chip memory, so much of their design concentrates on orchestrating data movement among off-chip memory, the PPE's cache, and the SPEs' local stores. They divide the FFT into three stages and synchronize the SPEs using mailboxes on stage boundaries.

5.6 Conclusions

We described a compiler for the SHIM concurrent language that generates code for the Cell processor. While not an aggressive optimizing compiler, it removes much of the drudgery in programming the Cell in C, which requires extensive library calls for starting threads, careful memory alignment of data if it is to be transferred between processors, and many other nuisances.

The SHIM language presents a scheduling-independent model to the programmer, i.e., relative task execution rates never affects the function computed by the program. This, too, greatly simplifies the programming task because there is no danger of introducing races or other nondeterministic behavior.

Unfortunately, our compiler does not solve a main challenge of parallel programming: creating well-balanced parallel algorithms. For example, the sequential portion of our FFT was able to keep six SPEs fed, leading to near-ideal speedups; the sequential portion of the JPEG decoder was substantial and became the bottleneck.

Our compiler does help to identify bottlenecks: it provides a mechanism for capturing precise timing traces using the Cell's precision timers. This gives a precise summary of when and how long each SPE is blocked waiting for communication, which can illustrate poorly balanced computational loads.

The Cell processor is an intriguing architecture that is representative of architectures we expect to find in many future embedded systems. While it has many idiosyncrasies, our work shows that it is possible to map a higher-level parallel programming model onto it and obtain reasonable performance.

Chapter 6

SHIM as a Library

5 In the previous chapters, we described the SHIM language and its code generation techniques for different architectures. In this chapter, we wish to evaluate the SHIM model as a library rather than a new programming language. We present a deterministic concurrent communication library for an existing multithreaded language. We implemented the SHIM communication model in the Haskell functional language, which supports asynchronous communication and transactional memory. The SHIM model uses multiway rendezvous to guarantee determinism.

Haskell actually supports several concurrency mechanisms, but does not guarantee functional determinism. We chose Haskell because it has a fairly mature STM implementation, carefully controlled side effects, and lightweight user-mode scheduled threads. We were also curious about whether our SHIM model, which we proposed previously for an imperative setting, would translate well to a functional language.

We implemented two versions of our library: one that uses mailboxes for interthread communication and one that uses software transactional memory. Experimentally, we found that mailboxes are more efficient for implementing the multiway rendezvous mechanism, especially for large numbers of processes. We also found our library easier to code using mailboxes.

After reviewing some related work, and Haskell's concurrency model, we describe our library and its implementation in Section 6.4 and present a series of experiments with our library on an eight-processor machine in Section 6.5.

6.1 SHIM as a Library Versus a Language

The SHIM model provides functional determinacy irrespective of being implemented as a language or a library, so an obvious question is which is preferred.

We present the library approach in this thesis. A library can leverage existing language facilities (editors, compilers, etc.) but does not provide guarantees about its misuse. A program that uses our library is functionally deterministic if it only uses our library for interthread communication, but there is nothing to prevent other mechanisms from being used.

The SHIM language does not provide any other interthread communication mechanism, guaranteeing determinism. However, the SHIM language and compiler are not as mature or feature rich as Haskell, the implementation vehicle for our library.

6.2 Related Work

The advent of mainstream multicore processes has emphasized the challenges of concurrent programming. Techniques ranging ranging from new concurrent languages to new concurrent libraries for existing languages are being investigated. Cω [18] is an example of a new research language, which provides join patterns in the form of chords that synchronize the arrival of data on multiple channels to atomically capture and bind values that are used by a handler function (such chords are also easy to implement in an STM setting). This pattern can capture many kinds of concurrency mechanisms, including rendezvous and actors, but it is nondeterministic and suffers from all the debugging challenges the SHIM model avoids.

Cilk [19] is another C-based language designed for multithreaded parallel programming that exploits asynchronous parallelism. It provides deterministic constructs to the programmer, but it is the programmer's responsibility to use them properly; the compiler does not guarantee determinism. This is one of the major differences between SHIM and Cilk. Cilk focuses on the runtime system, which estimates the complexities of program parts.

We built our library in Haskell, a functional language with support for concurrency [68]. Its concurrency mechanisms are not deterministic; our library provides a deterministic layer over them. Experimentally, we find such layering does not impose a significant performance penalty.

Our library resembles that of Scholz [107], which also provides an existing concurrency model in Haskell. Unlike Scholz, however, we implement our mechanisms atop the existing concurrency facilities in Haskell [68] and insist on functional determinism.

6.3 Concurrency in Haskell

We built our deterministic communication library atop Haskell's concurrency primitives. The most basic is *forkIO*, which creates an explicit thread and does not wait for its evaluation to complete before proceeding.

We implemented two versions of our library: one using mailboxes [68] for interthread communication, the other using software transactional memory [59; 46]. On a mailbox, *takeMVar* and *readMVar* perform destructive and non-destructive reads; *putMVar* performs a blocking write. Similarly, within the scope of an *atomically* statement, *readTVar* and *writeTVar* read and write transactional variables. Other threads always perceive the actions within an *atomically* block as executing atomically.

```
sampleMailbox
 = do
  m <- newEmptyMVar -- Create a new mailbox
  n <- newEmptyMVar
  forkIO (putMVar m (5::Int)) -- thread writes 5 to m
  forkIO (do
            c <- takeMVar m -- thread reads m
            putMVar n (c+1)) -- write to n
  result <- takeMVar n -- block for result
  return result
```

Figure 6.1: Using mailboxes in Haskell. One thread writes to mailbox *m*, a second reads *m*, adds one, and writes to mailbox *n*. The outer thread blocks on *n* to read the result.

The Haskell code in Figure 6.1 creates a mailbox `m` and forks two threads. The first thread puts the value 5 into `m` and the second thread takes the value from the mailbox `m`, adds one to it, and puts it in mailbox `n`.

Haskell's software transactional memory mechanisms [59; 46] are another way to manage communication among concurrent threads. In STM, threads can communicate or manipulate shared variables by reading or writing transactional variables. Statements within an *atomically* block are guaranteed to run atomically with respect to all other concurrent threads. A transaction can block on a *retry* statement. The transaction is rerun when one of the transaction variables changes.

The code in Figure 6.2 reads *c* and updates it if its value is not -1. The *atomically* guarantees the read and write appear atomic to other threads. The thread blocks while *c* is -1, meaning no other thread has written to it.

```
sampleSTM c
 = atomically (do
     value <- readTVar c
     if value == -1 then
          retry -- not written yet
       else writeTVar c (value + 1))
```

Figure 6.2: A Haskell program using STM. This updates the shared (“transactional”) variable c when it is not -1, otherwise blocks on c.

6.4 Our Concurrency Library

In this section, we present our SHIM-like concurrency library and its implementation. Our goal is to provide an efficient high-level abstraction for coding parallel algorithms that guarantees functional determinism. As described above, Haskell already has a variety of concurrency primitives (mailboxes and STM), but none guarantee determinism. Our hypothesis is that determinism can be provided in an efficient, easy-to-code way.

```
produce [c]
 = do
      val <- produceData
      dSend c val
      if val == -1 then -- End of data
           return ()
        else
           produce [c]

consume [c]
 = do
      val <- dRecv c
      if val == -1 then -- End of data
          return ()
        else
          do consumeData val
             consume [c]

producerConsumer
 = do
     c <- newChannel
     (_,_) <- dPar produce [c]
                   consume [c]
     return ()
```

Figure 6.3: A simple producer-consumer system using our library

6.4.1 Our Library's API

Our library provides channels with multi-way rendezvous and a facility for spawning concurrent threads that communicate among themselves through channels.

Figure 6.3 illustrates the use of our API. The *producerConsumer* function uses *newChannel* to create a new channel *c* and passes it to the *produce* and *consume* functions, which *dPar* runs in parallel. The producer sends data to the consumer, which consumes it while the producer is computing the next iteration. For communication costs not to dominate, evaluating *produceData* and *consumeData* should be relatively costly. Depending on which runs first, either the *dSend* of the producer waits for *dRecv* of the consumer or vice-versa, after which point both proceed with their execution to the next iteration.

Such a mechanism is also convenient for pipelines, such as Figure 6.4. The four functions run in parallel. The first feeds data to *pipelineStage1*, which receives it as *val1*, processes it and sends the processed data *val2* to *pipelineStage2* through channel *c2*. *PipelineStage2* acts similarly, sending its output to *outputFromPipeline* through *c3*.

Figure 6.5 shows the formal interface to our library. *newChannel* creates a new rendezvous channel. *dPar* takes four arguments: the first two are the first function to run and the list of channels passed to it; the last two are the second function and its channel connections. *dSend* takes two parameters: the channel and the value to be communicated. *dRecv* takes the channel as argument and returns the value in the channel.

6.4.2 Deadlocks and Other Problems

While our library guarantees functional determinism, it does not prevent deadlocks. For example, our library deadlocks when multiple threads call *dSend* on the same channel (a channel may only have one writer). While this could be detected, other deadlocks are more difficult to detect. If no sender ever rendezvous, the readers will block indefinitely.

Two threads that attempt to communicate on shared channels in different orders will deadlock. For example,

```
dSend c1 value            dSend c2 value
dRecv c2                  dRecv c1
```

will deadlock because the left thread is waiting for the right to rendezvous on *c1*, while the right is waiting for the left to rendezvous on *c2*. Such a deadlock is deterministic: the scheduler cannot make it disappear.

```
inputToPipeline [c1]
 = do
      val1 <- getVal
      dSend c1 val1
      inputToPipeline [c1]

pipelineStage1 [c1, c2]
 = do
      val1 <- dRecv c1
      val2 <- process1 val1
      dSend c2 val2
      pipelineStage1 [c1, c2]

pipelineStage2 [c2, c3]
 = do
     val2 <- dRecv c2
     val3 <- process2 val2
     dSend c3 val3
     pipelineStage2 [c2, c3]

outputFromPipeline [c3]
 = do
     val3 <- dRecv c3
     putStrLn (show val3)
     outputFromPipeline [c3]

pipelineMain
 = do
     c1 <- newChannel
     c2 <- newChannel
     c3 <- newChannel
     let dPar2 fun1 clist1 fun2 clist2 clist
          = dPar fun1 clist1 fun2 clist2
     let forkFunc1 = dPar2 inputToPipeline [c1]
                           pipelineStage1 [c1, c2]
     let forkFunc2 = dPar2 pipelineStage2 [c2,c3]
                           outputFromPipeline [c3]
     dPar forkFunc1 [c1, c2]
          forkFunc2 [c2, c3]
     return ()
```

Figure 6.4: A two-stage pipeline in our library

```
newChannel :: IO (Channel a)
dPar :: ([Channel a] -> IO b) ->
        [Channel a] ->
        ([Channel a] -> IO c) ->
        [Channel a] -> IO (b,c)
dSend :: Channel a -> a -> IO ()
dRecv :: Channel a -> IO a
```

Figure 6.5: The interface to our concurrency library. *newChannel* creates a new channel; *dPar* forks two threads and waits for them to terminate; *dSend* rendezvous on a channel and sends a value; and *dRecv* rendezvous and receives a value.

6.4.3 An STM Implementation

One implementation of our library uses Haskell's facilities for Software Transactional Memory (STM) [59]. Our goal was to see how difficult it would be to code and how efficient it would be for multi-way rendezvous. We describe the implementation below and defer experimental results to Section 6.5.

```
data Channel a = Channel {
      connections :: TVar Int,
      waitingReaders :: TVar Int,
      written :: TVar Bool,
      allReadsDone :: TVar Bool,
      val :: TVar (Maybe a)
    }
```

Figure 6.6: The channel type (STM)

```
newChannel
  = do
       connectionsT <- atomically $ newTVar 1
       waitingReadersT <- atomically $ newTVar 0
       writtenT <- atomically $ newTVar False
       allReadsDoneT <- atomically $ newTVar False
       valT <- atomically $ newTVar Nothing
       return (Channel connectionsT waitingReadersT
                       writtenT allReadsDoneT valT)
```

Figure 6.7: Creating a new channel (STM)

Figure 6.6 shows the collection of transactional variables used to represent a channel. The type variable *a* makes it polymorphic, *connections* tracks the number of threads that must rendezvous to perform the communication (it is adjusted by threads starting and terminating), *val* holds the data being communicated, *waitingReaders* tracks the number of threads that have blocked trying to read from the channel, *written* indicates whether the writer has written the data, and *allReadsDone* indicates when the last blocked reader has unblocked itself.

6.4.4 Forking parallel threads

```haskell
dPar func1 v1 func2 v2 = do
  done <- newEmptyMVar
  let common =
        intersectBy
          (\ x y -> (val x) == (val y)) v1 v2
  atomically (do
    apply (\ c -> do
        nt <- readTVar (connections c)
        writeTVar (connections c) (nt + 1)
      ) common)
  forkIO (do
    res <- func1 v1 -- Run func1 in child
    putMVar done res) -- Save result
  res2 <- func2 v2 -- Run func2 in parent
  res1 <- takeMVar done -- Get func1 result
  atomically (do
    apply (\ c -> do
        nt <- readTVar (connections c)
        writeTVar (connections c) (nt - 1)
      ) common)
  return (res1, res2)

apply func [] = return ()
apply func (hd:tl) = do func hd ; apply func tl
```

Figure 6.8: Our implementation of dPar

Figure 6.8 shows our implementation of *dPar* for STM. It creates a new MVar to hold the result from the child thread, then determines which channels are shared (*v1* and *v2* holds their names) and atomically increases their *connections*.

To evaluate the two functions, the parent forks a thread. The child thread evaluates *func2* and then writes the result into the mailbox. Meanwhile, the parent evaluates *func1*, waits for the child to report its result, atomically decreases the connection count on shared channels, and finally returns the results from *func1* and *func2*.

Figure 6.9 illustrates how *connections* evolves as threads fork and terminate. In Figure 6.9(a), F0 has spawned F1 and F2, increasing *connections* to 2. In (b), F2 has spawned F3 and F4, increasing *connections* to 3. Finally, in (c), F3 and F4

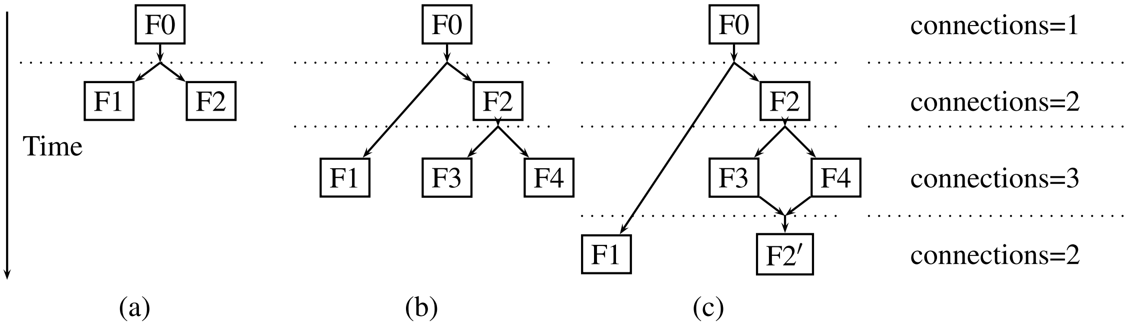


Figure 6.9: The effects on *connections* when (a) main function F0 calls *dPar F1 [c] F2 [c]*, then (b) F2 calls *dPar F3 [c] F4 [c]*, and (c) when F3 and F4 terminate.

have terminated, reducing *connections* to 2.

Note that this only happens when F0, . . . , F4 are all connected to channel *c*. If a thread was not connected, spawning it would not require the number of connections to change. This is what the computation of *common* in Figure 6.8 accomplishes by looking for channels passed to both threads being started.

6.4.5 Deterministic send and receive

Multi-way rendezvous is a three-phase process: wait for all peers to rendezvous, transfer data, and wait for all peers to complete the communication. Our library supports single-writer, multiple-reader channels, so if n_c is the number of threads connected to channel c, a writer waits for $n_c - 1$ readers; a reader waits for one writer and $n_c - 2$ other readers. We describe how to maintain n_c in the next section.

Figure 6.10 illustrates a scenario with two readers and a writer. Threads T1 and T3, call dRecv and dSend respectively. T1 and T3 wait for thread T2 to communicate. Once T2 calls dRecv, the three threads rendezvous and exchange data and continue with their individual execution.

Figure 6.11 shows our implementation of *dSend* using STM. It first waits for $n_c - 1$ readers to rendezvous, invoking *retry* to delay. Once they have, it atomically writes the value to send in *val* and resets the number of waiting readers, the *written* flag, and the *allReadsDone* flag. Finally, it waits for all the last receiver to set *allReadsDone*.

Figure 6.12 is the complementary process. It first increments *waitingReaders*, then waits for the *written* flag to be set by *dSend*. Once it has, it reads *val*—the data being communicated, increases *waitingReaders*, and sees if it was the last one. If it was, it resets *waitingReaders*, *allReadsDone*, and *written*, thereby releasing all the readers (including itself) and the writer. Otherwise, it waits for another reader

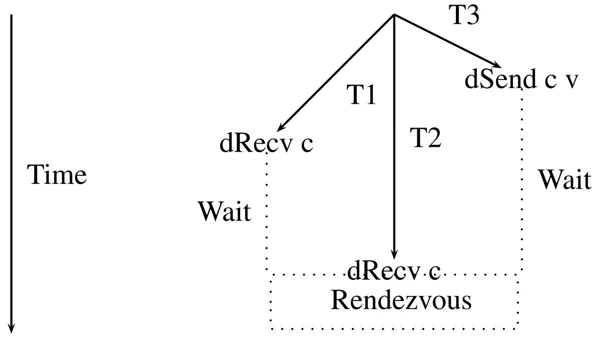


Figure 6.10: A rendezvous among two readers and one writer

```
dSend c value = do
  atomically (do
    wr <- readTVar (waitingReaders c)
    connections <- readTVar (connections c)
    if wr < connections - 1 then retry else (do
      writeTVar (val c) (Just value)
      writeTVar (waitingReaders c) 0
      writeTVar (written c) True
      writeTVar (allReadsDone c) False))
  atomically (do
    ard <- readTVar (allReadsDone c)
    if ard == False then retry else return ())
```

Figure 6.11: dSend (STM)

```
dRecv c = do
  atomically (do
    wr <- readTVar (waitingReaders c)
    writeTVar (waitingReaders c) (wr + 1)
    return ())
  v <- atomically (do
    w <- readTVar (written c)
    if w == False then retry else (do
      Just v <- readTVar (val c)
      wr <- readTVar (waitingReaders c)
      writeTVar (waitingReaders c) (wr + 1)
      nc <- readTVar (connections c)
      -- If last reader to read
      when (wr + 1 == nc - 1) (do
        writeTVar (waitingReaders c) 0
        writeTVar (allReadsDone c) True
        writeTVar (written c) False)
      return v))
  atomically (do
    ard <- readTVar (allReadsDone c)
    if ard == False then retry else return () )
  return v
```

Figure 6.12: dRecv (STM)

to set *allReadsDone*.

6.4.6 A Mailbox Implementation

For comparison, we also implemented our multiway rendezvous library using Haskell's mailboxes [68].

```
data Channel a = Channel {
      mVal :: MVar a,
      mVarCount :: MVar Int,
      mVarBegin :: MVar (),
      mVarEnd :: MVar ()
    }
```

Figure 6.13: The channel type (Mailboxes)

Figure 6.13 shows the *Channel* structure used to represent the channel. Field *mVal* holds the data, *mVarCount* holds the number of connections to this channel, and *mVarBegin* and *mVarEnd* are synchronization variables.

Figure 6.16 shows the *dRecv* procedure. A receiver sends a signal to the sender indicating it has arrived, then the receiver waits for the value from the sender. Once

```
newChannel
  = do
       mVal <- newEmptyMVar
       mVarCount <- newMVar 1
       mVarBegin <- newEmptyMVar
       mVarEnd <- newEmptyMVar
       return (Channel mVal mVarCount
                     mVarBegin mVarEnd)
```

Figure 6.14: newChannel (Mailboxes)

```
dSend (Channel mVar mVarCount
              mVarBegin mVarEnd) val = do
  waitForRecvrsToArrive mVarCount mVarBegin 1
  -- Wait for every receiver to send a sync.
  n <- readMVar mVarCount
  sendValueToRecvrs mVar val (n-1)
  putMVar mVar val
  takeMVar mVar
  signalRecvrs mVarEnd (n-1)

sendValueToRecvrs mVar value count = do
    if (count == 0) then
         return ()
       else do putMVar mVar value
               sendValueToRecvrs mVar
                       value (count - 1)
               return ()

waitForRecvrsToArrive mVarCount mVarBegin i
 = do
    count <- readMVar mVarCount
    if (count == i) then return ()
     else do
          takeMVar mVarBegin
          waitForRecvrsToArrive mVarCount
                              mVarBegin (i+1)

signalRecvrs mVarEnd count
 = do if (count == 0)
        then return ()
       else do putMVar mVarEnd ()
               signalRecvrs mVarEnd (count-1)
```

Figure 6.15: dSend (Mailboxes)

all receivers have read the value, the sender signals an end, after which dRecv returns with the value.

The *dSend* procedure (Figure 6.15) waits for all receivers, then performs a *putMVar* on the value once per receiver. To ensure the last receiver has read, it does a redundant *putMVar* and *takeMVar*. Finally, once all receivers have read the value, it signals the receivers to continue execution. *WaitForRecvrsToArrive* waits for every receiver to send a sync indicating it has arrived. *SignalRecvrs* signals the end by informing each receiver the rendezvous is complete.

```
dRecv (Channel mVar mVarCount
          mVarBegin mVarEnd)
 = do
    putMVar mVarBegin () -- Inform sender
    value <- takeMVar mVar -- Wait for sender
    takeMVar mVarEnd -- Wait for sender end
    return value
```

Figure 6.16: dRecv (Mailboxes)

6.5 Experimental Results

To test the practicality and efficiency of our library, we created a variety of programs that used it and timed them.

6.5.1 STM Versus Mailbox Rendezvous

As a basic test of efficiency, we had our library rendezvous 100 000 times among various numbers of threads on a two-processor machine (a 500 MB, 1.6 GHz Intel Core 2 Duo running Windows XP) and measured the time. Table 6.1 lists the results.

Mailboxes appear to be more efficient for our application, especially when large numbers of threads rendezvous. We believe this may be fundamental to the STM approach, in which threads continue to execute even if there is a conflict. Only at the end of the transaction is conflict checked and rolled back if needed. In the case of a multiway rendezvous, many threads will conflict and have to be rolled back. Mailboxes are more efficient here because they check for conflicts earlier.

The STM method also requires more memory to hold the information for a roll back. Again, mailboxes have less overhead because they do not need this information.

Threads	Time to Rendezvous		Speedup
	STM	Mailbox	(STMMailbox)
2	0.11 ms	0.07 ms	1.6
3	0.14	0.08	1.8
4	0.17	0.14	1.2
5	0.21	0.16	1.3
6	0.28	0.17	1.6
7	0.31	0.21	1.5
8	0.37	0.23	1.6
9	0.42	0.27	1.6
10	0.47	0.28	1.7
100	6.4	1.8	3.5
200	35	6.7	5.2
400	110	14	7.9
800	300	34	8.9

Table 6.1: Time to rendezvous for STM and Mailbox implementations

The STM method is more complicated. Unlike mailboxes, which only require a mutual exclusion object, a flag, and the data to be transferred, STM requires managing information to identify conflicts and roll back transactions.

However, the ratio of communication to computation is the most critical aspect of application performance. For a computationally intensive application, a 50% increase in communication time hardly matters.

6.5.2 Examples Without Rendezvous

These examples only call *dPar* and do not use *dSend* or *dRecv*. Our goal here is to compare our library with Haskell's existing par-seq facility, which we feel presents an awkward programming interface [103].

```
fib n | n <= 1 = 1
      | otherwise =
        par res1 (pseq res2 (res1 + res2 + 1))
        where res1 = fib (n - 1)
              res2 = fib (n - 2)
```

Figure 6.17: Calculating Fibonacci numbers with Haskell's par-seq

Haskell's par-seq constructs can be used to emulate our *dPar*. The following are semantically equivalent

$$dpar\ M\ \texttt{[]}\ N\ \texttt{[]} \leftrightarrow \texttt{(}par\ M\ \texttt{(}pseq\ N\ \texttt{(}M\texttt{,}\ N\texttt{)))}$$

but the *par* does not guarantee *M* and *N* are executed in parallel because Haskell uses lazy evaluation. Nevertheless, we find the par-seq method can run faster than our *dPar*.

Using par-seq is subtle, illustrated by Figure 6.17. While both *par* and *pseq* only return the value of their second argument, the meaning of *m1 par m2* is "start the calculation of *m1* for speculative evaluation and then go onto evaluate *m2*." This is useful when *m1* is a subexpression of *m2* so *m1* may be evaluated in parallel with the body of *m2*. Conversely, *pseq* makes sure its first argument is evaluated before evaluating its second. In this example, the *pseq* guarantees that *fib* (*n*-*2*) is evaluated before *fib* (*n*-*1*), which can use *fib* (*n*-*2*).

We find this mechanism subtle and difficult to control. It provides weak control over the scheduling of computation—a complex issue for a lazy language like Haskell made all the more tricky by parallelism. We believe providing users with easy-to-use mechanisms to control scheduling is necessary for achieving high performance; expecting the compiler or runtime system to make the best choices seems unrealistic.

We ran these and all later experiments on an 8-processor Intel machine containing two 5300-series 1.6 GHz quad processors, 2 GB of RAM, and running Windows NT Server.

6.5.3 Maximum element in a list

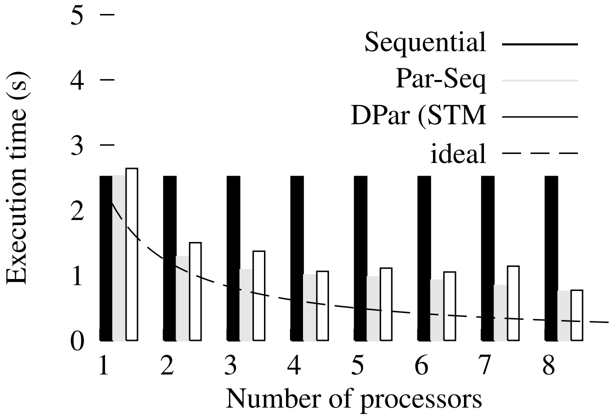


Figure 6.18: Maximum Element in a List

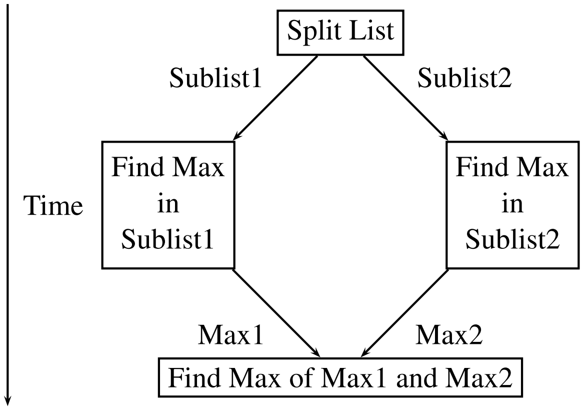


Figure 6.19: Structure of Maximum Finder

Figure 6.18 shows the execution times for a program that uses a linear search to find the maximum element in a 400 000-element list. The program, whose behavior is shown in Figure 6.19, splits a list into pieces, one per thread, finds the maximum of each piece, and finally finds the maximum of the pieces. We compared a sequential implementation, one that uses Haskell's par-seq constructs, and one that uses our *dPar* to the ideal speedup of the sequential implementation.

Figure 6.18 shows the par-seq implementation is slightly more efficient, although both implementations fall short of the ideal $1/n$ speedup on more than two processors.

6.5.4 Boolean Satisfiability

Figure 6.20 shows the execution times of a simple Boolean SAT solver implemented sequentially, using par-seq, and with our *dPar*. We ran it on an unsatisfiable problem with 600 variables and 2 500 clauses. Figure 6.21 shows the structure of our approach: we pick an unassigned variable and spawn two threads that check whether the expression can be satisfied if the variable is true or false. Because of our demand for determinism, we do not asynchronously terminate all the threads when one finds the expression has been satisfied. Our algorithm is also primitive in the sense that it does not do any online learning.

Again, we find our *dPar* has more overhead than Haskell's par-seq. Also, this algorithm does not appear to benefit from more than four processors, which we attribute in part to Haskell's single-threaded garbage collector.

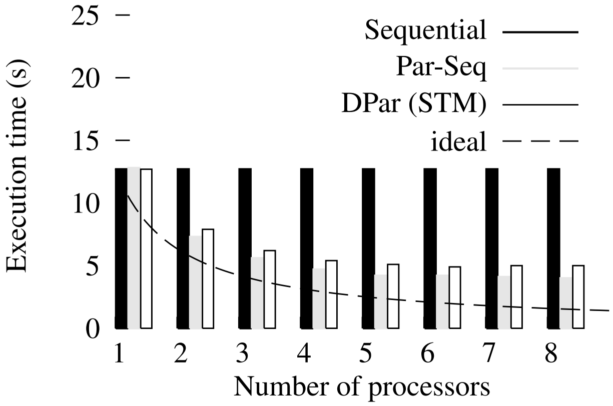


Figure 6.20: Boolean Satisfiability

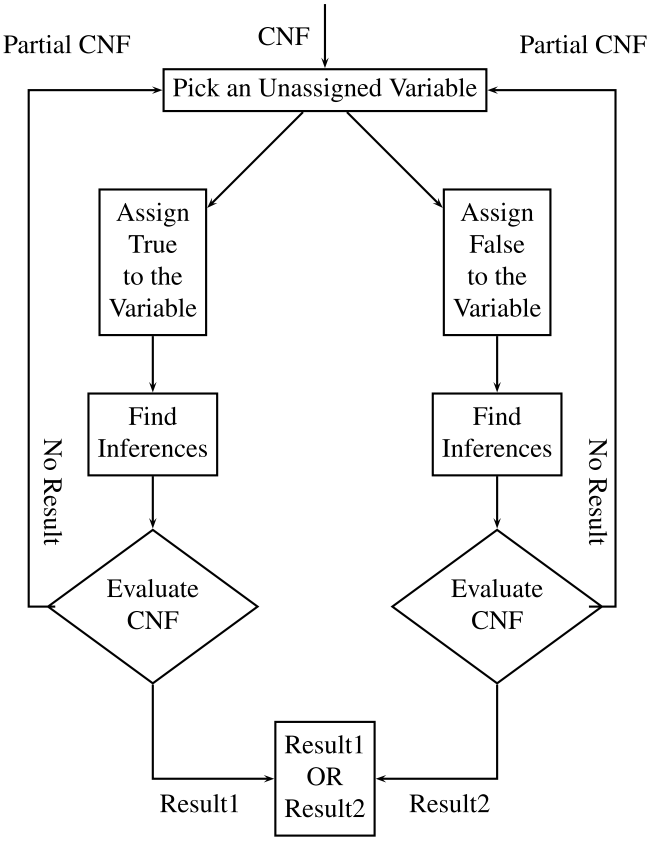


Figure 6.21: Structure of the SAT Solver

6.5.5 Examples With Rendezvous

Here we consider algorithms that use rendezvous communication among threads. The comparisons are to purely sequential implementations of the same algorithm.

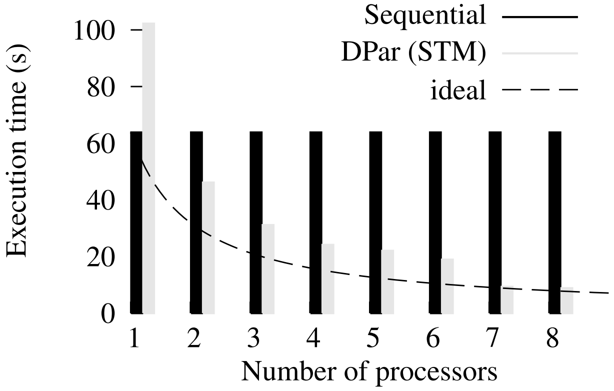


Figure 6.22: Times for Linear Search

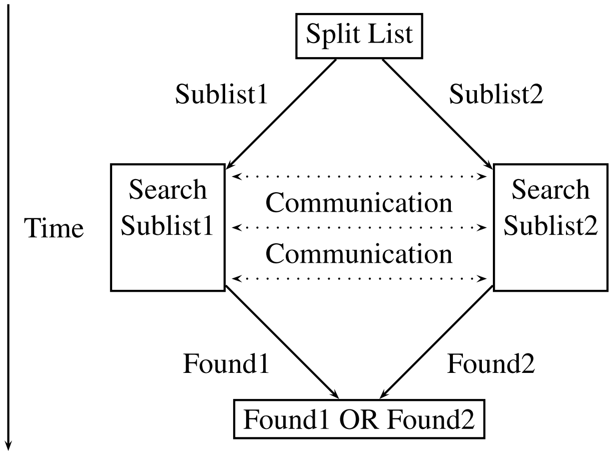


Figure 6.23: Linear Search Structure

6.5.6 Linear Search

Figure 6.22 shows the execution times of a linear search program that uses rendezvous communication to find a key in a 420 000-element list (we put it in the

390 000th position). Unlike the maximum element problem, linear search generally does not need to scan the list completely, so the algorithm should have a way of terminating early.

Requiring determinism precludes the obvious solution of scanning *n* list fragments in parallel and terminating immediately when the key is found. This constitutes a race if the key appears more than once, since the relative execution rates of the threads affect which copy was reported.

Our implementation takes the approach shown in Figure 6.23: the list is broken into *n* fragments and passed to parallel threads. However, rather than asynchronously terminating all the threads when the key is found, instead all the threads rendezvous at a prearranged interval to check whether any have found the key. All threads proceed if the key is not found or terminate and negotiate which copy is reported if one has been found.

This technique trades off communication frequency and the amount of extra work likely to be done. Infrequent communication means less overhead, but it also makes it more likely the threads will waste time after the key is found. Frequent communication exchanges overhead for punctuality. We did not have time to explore this trade-off.

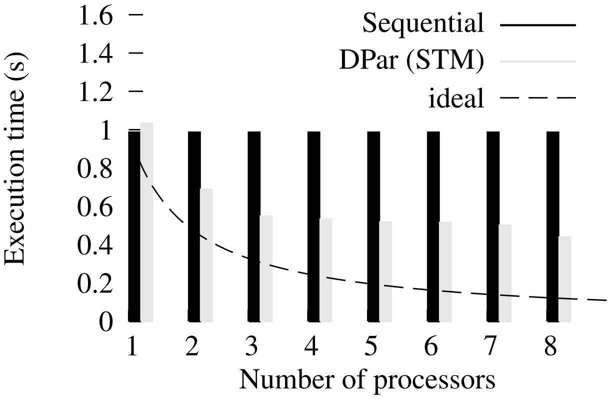


Figure 6.24: Systolic Filter

6.5.7 Systolic Filter and Histogram

Figure 6.24 shows the execution times of a Systolic 1-D filter running on 50 000 samples. Each thread run by the filter can independently process a chunk of the input in parallel with other threads following the structure in Figure 6.26. Because of determinism, jobs are distributed and collected from the worker threads in a

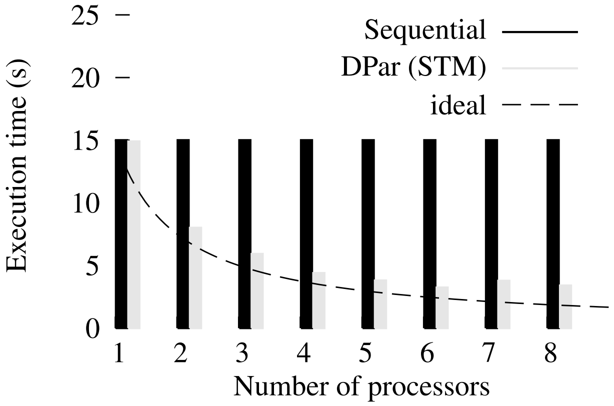


Figure 6.25: RGB Histogram

round-robin order.

Figure 6.25 shows the execution time of a similar algorithm: a histogram of RGB values in an image. We ran it on a 565 KB raster file.

6.6 Conclusions

While we found it was reasonable and fairly efficient to implement a deterministic concurrency library based on multi-way rendezvous, our efforts did raise a few issues.

We found that the performance of our library was slightly lower than that of Haskell's built-in par-seq mechanism. We suspect this is from additional layers of abstraction between our library calls and the par-seq mechanism. Despite this, we believe our library provides a nicer abstraction because it makes communication and synchronization explicit and therefore makes an easier optimization target, but this is difficult to quantify.

While we were successful at implementing the library using both Mailboxes and software transactional memory (STM), we are happier with the mailbox-based implementation because it is both faster and easier to program and understand. While it is clearly possible to wait to synchronize with peers in STM, coding it seems needlessly awkward. We also observed STM increased synchronization overhead by at least 50%, although this is not prohibitive.

Our experiences do provide insight for the library vs. language debate. While the library approach has the advantage of leveraging features of the host language, we encountered a number of problems that made the library difficult to implement and use.

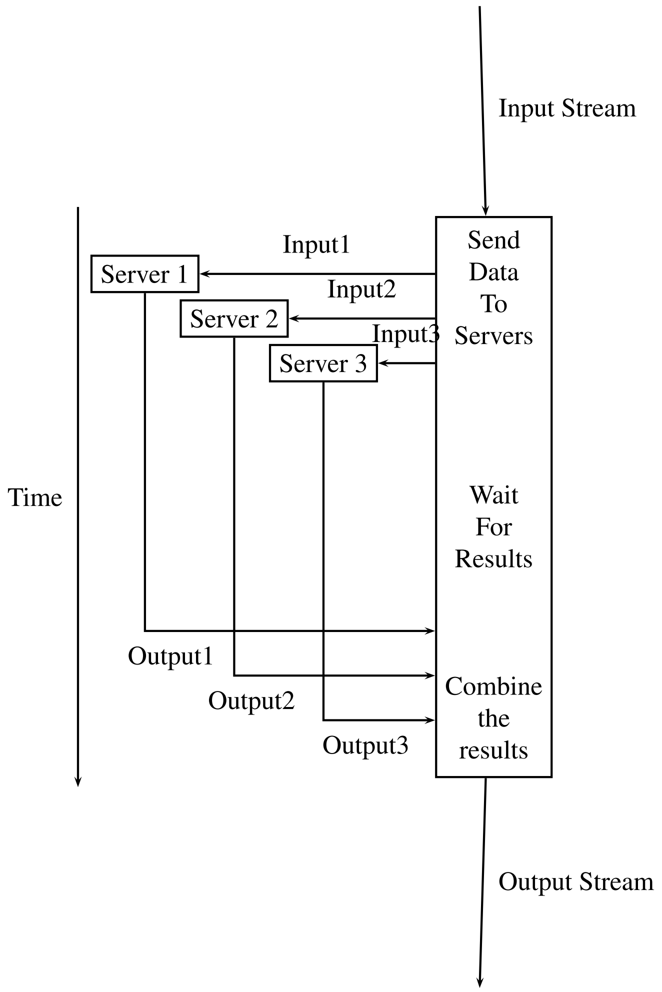


Figure 6.26: Server Programming Style used by Histogram and Systolic Filter

Unlike C, Haskell does not allow its type system to be circumvented. This avoids more runtime errors but makes building really polymorphic things harder. We would like a *dPar* that spawns an arbitrary number of threads, each of which is connected to an arbitrary number and type of channels. Such flexibility is difficult to express in a library. We settled on spawning only two threads at a time (*n*-way forks can be recovered by nesting) and not checking the number of channels, thus deferring certain errors to runtime. Haskell probably allows a more flexible interface, but the code can become very obscure.

The type system in C is easy to circumvent and C allows functions with a variable number of parameters, so a C implementation of our library might have a cleaner API. However, going around the type system opens the door to runtime type errors, e.g., trying to pass a string through a channel intended for floating-point numbers.

We believe our library presents an easier, less error-prone programming model than either mailboxes or STM, but this is hard to prove. Anecdotally, we found it easier to debug, especially deadlocks, which were reproducible. Furthermore, it seems easier to reason about explicit synchronization instead of explicitly using *retry* in the STM setting.

Part III

Deadlock-freedom

Outline

Although we achieved determinism in Part II, we still suffer from the deadlock problem. SHIM is not immune to deadlocks, but they are simpler to manage because of SHIM's scheduling independence. Deadlocks in SHIM cannot occur because of race conditions. For example, because SHIM does not have races, there are no race-induced deadlocks, such as the "grab locks in opposite order" deadlock race present in many other languages. A key hypothesis of the SHIM model has been that scheduling independence should be a property of any practical concurrent language because it greatly simplifies reasoning about a program, both by the programmer and by automated tools. This part on deadlock detection reinforces this key point.

Chapter 7

Deadlock Detection for SHIM using a Synchronous Model Checker

As discussed in Chapter 3, the SHIM concurrent language guarantees the absence of data races by eschewing shared memory, but a SHIM program may still deadlock if a program violates the communication protocol.

```
void main() {
  chan int a, b;
  {
    // Task 1
    next a = 5; // Deadlocks here
    next b = 10;

  } par {

    // Task 2
    next b; // Deadlocks here
    next a;
  }
}
```

Figure 7.1: A SHIM program that deadlocks

In Figure 7.1, the two tasks also attempt to communicate, but task 1 attempts to synchronize on *a* first, then *b*, while task 2 expects to synchronize on *b* first. This is a deadlock—each task will wait indefinitely for the other.

In this chapter, we present a model-checking based static deadlock detection technique for the SHIM language. Although SHIM is asynchronous, its semantics allow us to model it synchronously without losing precision, greatly reducing the state space that must be explored. This plus the obvious division between control and data in SHIM programs makes it easy to construct concise abstractions.

A central goal of our work was to confirm that a careful choice of concurrent semantics simplifies the verification problem. In particular, while SHIM's semantics are asynchronous, they are constructed so that checking a (much simpler) synchronous abstraction remains sound. In particular, we do not need the power of a model checker such as Holtzmann's SPIN [63], which is designed to analyze an interleaving concurrency model.

The synchronous abstraction we use to check for deadlock is sound because of SHIM's scheduling independence: the choice of scheduling policy cannot affect the function of the program. In particular, a program either always or never deadlocks for a particular input—a scheduling choice cannot affect this. This means we are free to choose a particular scheduling policy without fear of missing or introducing deadlocks. Here, we choose a scheduling policy that amounts to a synchronous execution of a SHIM program. This greatly reduces the number of distinct states our model checker must consider, simplifying its job.

In this chapter, we propose a technique that builds a synchronous abstraction of a SHIM program and uses the NuSMV symbolic model checker to determine whether the model can deadlock. Because of SHIM's semantics, if the model does not deadlock, the program is guaranteed not to deadlock, but because we abstract the data values in the program, the converse is not true: a program may not deadlock when we report it does.

By design, SHIM is a finite-state language provided it has no unbounded recursion (Edwards and Zeng [52] show how to remove bounded recursion), which makes the deadlock problem decidable. Unfortunately, exact analysis of SHIM programs can be impossible in practice because of state space explosion; we build sound abstractions instead.

Our main contribution is an illustration of how efficient, synchronous model-checking techniques can be used to analyze an asynchronous system. The result is a practical static deadlock detector for a particular asynchronous language. While the theoretical equivalence of synchronous and asynchronous models has long been known, we know of few other tools that exploit it.

This work confirms that having designed SHIM's semantics to be scheduling independent makes the language much easier to analyze with automated tools (elsewhere, we have argued that scheduling independence helps designers understand programs [47]). Few other concurrent languages were designed with formal verification in mind.

After reviewing related work, we show how we abstract SHIM programs to make their deadlock properties easy to analyze. After that, we detail the generation of a NuSMV model and present experimental results that show our method is practical enough to run as part of a normal compilation flow. Overall, this suggests that careful language design can simplify the challenge of concurrent programming by making it easy to automatically check for certain problems.

7.1 Related Work

Avoiding deadlocks and data races in concurrent programs has been studied intensively; both static and dynamic techniques have been proposed. Detecting deadlocks in a running system is easy if global information is available. Distributed algorithms, such as Lee and Kim's [75], are more complicated, but computationally inexpensive. In this chapter, we focus on the harder, more interesting problem of detecting potential deadlocks before a system runs since this is when we want to correct them.

As we propose in this chapter, model checking is often used for static deadlock. Corbett [40] reviews a variety of static deadlock detection methods for concurrent Ada programs. He observes the main challenge is dealing with the state explosion from Ada's interleaved concurrency model. SHIM's scheduling-independent semantics sidesteps this problem. Taking a very different approach, Boyapati and Rinard [21] propose a static type system for an extended Java language based on programmer-declared ownership of each object. Their system guarantees objects are only accessed in a synchronized manner. SHIM guarantees unique ownership by simply prohibiting shared objects.

The interleaved concurrency model in most concurrent software environments is a challenge for model checkers. Many, such as SPIN [63], Improvsio [77], and Java PathFinder [133] use partial order reduction to reduce the number of states they much consider. While SHIM is asynchronous, its communication semantics do not require all possible interleavings to be considered, making the model checking problem much easier because we can check a synchronous model with far fewer states.

SHIM does not provide locks (although some of its implementations employ them) so it presents no danger of deadlock from bad locking policies. Hence lock-focused analysis, such as Bensalem et al. [13], which examines a single (non-deadlocking) trace for potential deadlock situations, is not applicable to SHIM.

7.2 Abstracting SHIM Programs

A sound abstraction is the central idea behind our deadlock detector for SHIM. A SHIM task alternates between computation and communication. Because tasks only interact when they communicate and never deadlock when they are computing, we abstract away the computation and assume a task always either communicates again or terminates, i.e., will never enter an infinite loop that never communicates. This is tantamount to assuming a schedule that perfectly balances the computation time of each process.

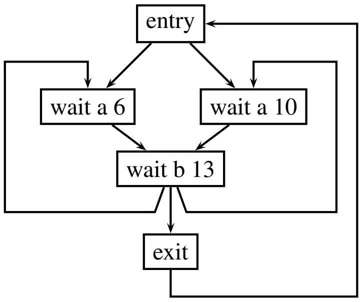
This assumption is optimistic in the sense that our tool may report a program is deadlock-free even if one of its tasks enters an infinite loop where it computes forever. However, checking for process termination can be done independently and can likely consider tasks in isolation. Answering the task termination question is outside the scope of this chapter.

We also abstract away the details of data manipulation and assume all branches of any conditional statement can always be taken at any time. This is a conservative assumption that may increase the number of states we must consider. As usual, by ignoring data, we leave open the possibility that two tasks may appear to deadlock but in fact stay synchronized because of the data they exchange, but we believe this abstraction is a reasonable one and furthermore believe that system that depend on such behavior are probably needlessly difficult for the programmer to understand. In Section 7.5, we show an example of working code for which our tool reports a deadlock and how to restructure it to avoid the deadlock report.

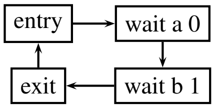
```
void main() {
  int i;
  chan int a, b;
  {
    for (i = 0 ; i < 100 ; i++) {
      if (i % 10)
        next a = 1;
      else
        next a = 0;
      next b = 10;
    }
  } par {
    next a;
    next b;
  }
}
```

Figure 7.2: A deadlock-free SHIM program with a loop, conditionals, and a task that terminates

```
main_1(chan int32 &a, chan int32 &b)
local int32 i
local int32 tmp0
local int32 tmp1
  i = 0
  goto continue
while:
  tmp1 = i % 10
  ifnot tmp1 goto else
  a = 1
  send a // 6
  goto endif
else:
  a = 0
  send a // 10
endif:
  b = 10
  send b // 13
  i = i + 1
continue:
  tmp0 = i < 100
  if tmp0 goto while
```



```
main_2(chan int32 a,
       chan int32 b)
  recv a // 0
  recv b // 1
```



```
main()
channel int32 a
channel int32 b
  main_1(a, b) : main_2(a, b);
```

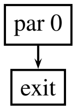


Figure 7.3: The IR and automata for the example in Figure 7.2. The compiler broke the *main* function into three tasks.

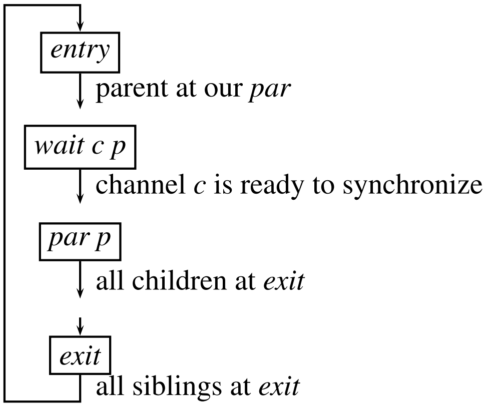


Figure 7.4: The four types of automaton states. Each has one *entry* and *exit*, but may have many *wait* and *par* states.

7.2.1 An Example

Consider the SHIM program in Figure 7.2. The *main* function starts two tasks that communicate through channels *a* and *b*. The first task communicates on channels *a* and *b* 100 times; the second task synchronizes on channels *a* and *b*, then terminates. This program does not deadlock because the communication patterns of the two tasks mesh properly. Note that SHIM semantics say that once a task terminates, it is no longer compelled to synchronize on the channels to which it is connected. So here, after the second task synchronizes on *b* and terminates, the first task talks to itself.

To abstract this program, our compiler begins by dismantling the SHIM program into a traditional, three-address-code-style IR (Figure 7.3). The main difference is that *par* constructs are dismantled into separate functions, here *main_1* and *main_2*, to ensure each function is sequential.

We assume the overall SHIM program is not recursive and remove statically bounded recursion using Edwards and Zeng [52]. We do not attempt to analyze recursive programs where the recursion cannot be bounded.

Next, we duplicate code to force each function to have a unique call site. While this has the potential for an exponential increase in code size, we did not observe it.

We remove trivial functions—those that do not attempt to synchronize. A function is trivial if it does not contain a *next* and all its children are trivial. Provided they terminate (an assumption we make), the behavior of such functions does not affect whether a SHIM program deadlocks. Fortunately, it appears that functions called in many places rarely contain communication (I/O functions are an exception), so the potential expansion from copying functions to ensure each has a unique call site rarely occurs in practice.

This preprocessing turns the call structure of the program into a tree, allowing us to statically enumerate all the tasks, the channels and their connections, and identify a unique parent and call site for each task (aside from the root).

Next, our tool creates an automaton that models the control and communication behavior for each dismantled function. Figure 7.3 shows automata and the code they model.

Each automaton consists of four kinds of states, shown in Figure 7.4. An automaton in its *entry* state waits for its parent to reach the *par* state at the automaton's call state. An automaton in its *exit* state waits for all its siblings to also be in their *exit* states. Each automaton (except the topmost one) starts in its *entry* state.

When an automaton enters a *par* state, it starts its children and waits for each of them to reach *exit* states. This is not a race because each child will advance from its *entry* state a cycle after the parent reaches the *par*. An automaton has one *par* state for each of its call sites. We label each with an integer that encodes the program counter.

Finally, *wait* states handle blocking communication. For an automaton to leave a *wait* state, all the running tasks that are connected to the channel (each *wait* corresponds to a unique channel) must also be at a *wait* for the channel. Note that an automaton may have more than one *wait* for the same channel; we label each with both the name of the channel and the program counter value at the corresponding *next*. The numbers 0, 1, 6, 10, and 13 in Figure 7.3 correspond to program counter values.

When we abstract a SHIM program, we ignore sequences of arithmetic operations; only conditionals, communication, and parallel function calls are preserved. Conditional branches, such as the test of *tmp1* in *main_1*, are modeled as nondeterministic choices.

We treat our automata as running synchronously, which amounts to imposing a particular

scheduling policy on the program. SHIM's scheduling independence guarantees that we do not affect the functional behavior of the program by doing this. And in particular, the program can deadlock under any schedule if and only if it can deadlock under this schedule. This is what makes our abstraction of the program sound.

We do not explicitly model the environment in which the program is running; instead, we assume it is part of the program being tested. A sensor or actuator could be modeled as an independent SHIM process that is always willing to communicate: a source or a sink. More complicated restrictions on environmental behavior would have to be modeled by more SHIM processes.

While we could build an explicit synchronous product automaton from the automata we build for each function, doing so would subject us to the state space explosion problem. Instead, we use a symbolic model checker that analyzes the product of these automata more efficiently.

```
MODULE main
DEFINE ready_a :=
  (main in {entry, exit} |
   main in {par_0} & (main_1 != exit & main_1 != entry |
                      main_2 != exit & main_2 != entry)) &
  main_1 in {entry, exit, wait_a_10, wait_a_6} &
  main_2 in {entry, exit, wait_a_0};

DEFINE ready_b :=
  (main in {entry, exit} |
   main in {par_0} & (main_1 != exit & main_1 != entry |
                      main_2 != exit & main_2 != entry)) &
  main_1 in {entry, exit, wait_b_13} &
  main_2 in {entry, exit, wait_b_1};

VAR main: {entry, exit, par_0};
ASSIGN init(main) := par_0;
  next(main) := case
  main = par_0 & main_2 = exit & main_1 = exit: exit;
  1: main;
esac;

VAR changed_main: boolean;
ASSIGN init(changed_main) := 1;
next(changed_main) := case
  main = par_0 & main_2 = exit & main_1 = exit: 1;
  1: 0;
esac;

VAR main_2: {entry, exit, wait_a_0, wait_b_1};
ASSIGN init(main_2) := entry;
next(main_2) := case
  main_2 = entry & main = par_0: wait_a_0;
  main_2 = wait_a_0 & ready_a: wait_b_1;
  main_2 = wait_b_1 & ready_b: exit;
  main_1 = exit & main_2 = exit: entry;
  1: main_2;
esac;
```

```
VAR changed_main_2: boolean;
ASSIGN init(changed_main_2) := 1;
next(changed_main_2) := case
  main_2 = entry & main = par_0: 1;
  main_2 = wait_a_0 & ready_a: 1;
  main_2 = wait_b_1 & ready_b: 1;
  main_1 = exit & main_2 = exit: 1;
  1: 0;
esac;

VAR main_1: {entry, exit, wait_a_10, wait_a_6, wait_b_13};
ASSIGN init(main_1) := entry;
next(main_1) := case
  main_1 = entry & main = par_0: {wait_a_10, wait_a_6, exit};
  main_1 = wait_a_6 & ready_a: wait_b_13;
  main_1 = wait_a_10 & ready_a: wait_b_13;
  main_1 = wait_b_13 & ready_b: {wait_a_10, wait_a_6, exit};
  main_1 = exit & main_2 = exit: entry;
  1: main_1;
esac;

VAR changed_main_1: boolean;
ASSIGN init(changed_main_1) := 1;
next(changed_main_1) := case
  main_1 = entry & main = par_0: 1;
  main_1 = wait_a_6 & ready_a: 1;
  main_1 = wait_a_10 & ready_a: 1;
  main_1 = wait_b_13 & ready_b: 1;
  main_1 = exit & main_2 = exit: 1;
  1: no;
esac;

SPEC AG(main != exit ->
          changed_main | changed_main_2 | changed_main_1)
SPEC EG(main != exit ->
          changed_main | changed_main_2 | changed_main_1)
```

Figure 7.5: NuSMV code for the program in Figure 7.2

7.3 Modeling Our Automata in NuSMV

To check whether our abstracted program (concurrently-running automata) deadlocks, we use the NuSMV [34] BDD- and SAT-based symbolic model checker. While it can analyze both synchronous and asynchronous finite-state systems, we only consider synchronous systems. The specifications to check can be expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL).

Using NuSMV involves supplying it with a model and a property of the model to be checked. We model each of our automata with two variables: one that represents the control state of the automaton and one that helps us determine when a deadlock has occurred. Figure 7.5 shows the code we generate for the three automata in Figure 7.3.

Translating a nondeterministic automaton into NuSMV is straightforward. We use the following template:

```
VAR state : {s1, s2, ... };
ASSIGN
  init(state) := s1;
  next(state) := case
    state = s1 & ... : {s2, s3, ...};
    state = s2 & ... : {s1, s3, ...};
    ...
    state = sn & ... : {s1, s2, ...};
    1 : state;
  esac;
```

For this automaton, *state* is a variable that can take on the symbolic values *s1*, *s2*, ... Each rule in the *case* statement is of the form *predicate*:*values*; and the predicates are prioritized by the order in which they appear.

Predicates are Boolean expressions over the state variables; values are sets of new values among which the model checker chooses nondeterministically. We model conditional branches in a SHIM program with nondeterministic choice. We generate one predicate/value pair for each state and start each predicate with a test for whether the machine is in that state. The final predicate/value pair is a default that holds the machine in its current state if it was not able to advance.

The NuSMV language has a rich expression syntax, but we only use Boolean connectives (& and |), comparison (=), and set inclusion (*in*).

For an automaton to leave its *entry* state, its parent must be in the *par* state for the automaton. By construction, both the parent automaton and the *par* state for a task is unique. For example, in Figure 7.5, the parent of *main_2* is *main*, and *main* calls *main_2* in the *par_0* state, so the rule for *main_2* to leave its *entry* state is

```
  main_2 = entry & main = par_0: wait_a_0;
```

since in *main_2*, the successor to the *entry* state is *wait_a_0*.

For an automaton to leave a *par* state, all the children at the call site must be in their *exit* states. In Figure 7.5, *main_1* and *main_2* are invoked by *main* in the *par_0* state, so the complete rule for *main* to leave its *par_0* state is

```
main = par_0 & main_2 = exit & main_1 = exit: exit;
```

since the successor of *par_0* in *main* is *exit*.

A state labeled *wait_c_p* represents a task waiting to synchronize on channel *c*. Since a task may wait on the same channel in many places, we also label it with a program counter value *p*. An automaton transitions from a *wait* state when all other automata that are compelled to rendezvous have also reached matching *wait* states.

The rules for when rendezvous can occur on a channel are complicated because tasks do not always run. When a task connected to channel *c* is running children (i.e., the task is blocked on *par* and its children are running), it passes its connection to its children. However, if all its children connected to *c* terminate (i.e., reach their *exit* states) the parent resumes responsibility for synchronization and effectively blocks communication on *c* until it reaches a *wait* on *c*.

For each channel *c*, we define a variable *ready_c* that is true when a rendezvous is possible on the channel. We form it by taking the logical *and* of the rendezvous condition for each task that can connect to the channel (we know statically which tasks may connect to a particular channel).

For each task on a channel *c*, the rendezvous condition is true if the task is in the *entry* state (when it has not been started), in the *exit* state (when it ran and terminated, but its siblings have not terminated), in a *wait* state for the channel, or in a *par* state when at least one of its children connected to *c* is still running (i.e., when the parent has not recovered its responsibility for the channel *c* from its children).

Figure 7.6 illustrates the rendezvous condition for a fairly complex set of tasks. Tasks 1, 2, 4, 5, 6, and 7 are leaves—they do not call other tasks. For each, the condition is that it be terminated or at a *wait* state for the channel.

Task 3 both synchronizes directly on *a* and invokes tasks 1 and 2. Its condition is that it be terminated, at its *wait* state for *a*, or that it be at its *par* state and at least one of task 1 or 2 be running.

Task 8 is the most complex. It synchronizes on *a* in two states (*wait_a_0* and *wait_a_2*) and has two *par* calls. At either of the *par* calls, at least one of its four children (tasks 3, 4, 6, and 7) must be running.

Note that since task 5 is not connected to channel *a*, its state is not considered.

```
{ // task 8
  next a;
  { // task 3
    next a;
      next a; // task 1
    par
      next a; // task 2
  } par { // task 4
    next a;
    next b;
  } par { // task 5
    next b;
  }
  next a;
    next a; // task 6
  par
    next a; // task 7
}
(t_8 in {enter, exit, wait_a_2, wait_a_0} |
 t_8 in {par_3, par_1} & (t_7 != exit & t_7 != enter |
                       t_6 != exit & t_6 != enter |
                       t_4 != exit & t_4 != enter |
                       t_3 != exit & t_3 != enter)) &
(t_3 in {enter, exit, wait_a_0} |
 t_3 in {par_1} & (t_2 != exit & t_2 != enter |
                  t_1 != exit & t_1 != enter)) &
t_7 in {enter, exit, wait_a_0} &
t_6 in {enter, exit, wait_a_0} &
t_4 in {enter, exit, wait_a_0} &
t_2 in {enter, exit, wait_a_0} &
t_1 in {enter, exit, wait_a_0}
```

Figure 7.6: A SHIM code fragment and the conditions for rendezvous on the *a* channel

Example	Lines	Channels	Tasks	Result	Runtime	Memory	States
Source-Sink	35	2	11	No Deadlock	0.2 s	3.9 MB	97
Pipeline	30	7	13	No Deadlock	0.1	2.0	95
Prime Number Sieve	35	51	45	No Deadlock	1.7	25.4	3.2×10^9
Berkeley	40	3	11	No Deadlock	0.2	7.2	139
FIR Filter	100	28	28	No Deadlock	0.4	13.4	4134
Bitonic Sort	130	65	167	No Deadlock	8.5	63.8	25
Framebuffer	220	11	12	No Deadlock	1.7	11.6	9593
JPEG Decoder	1020	7	15	May Deadlock	0.9	85.6	571
JPEG Decoder Modified	1025	7	15	No Deadlock	0.9	85.6	303

Table 7.1: Experimental results with NuSMV

7.4 Finding Deadlocks

We define a deadlock as a state in which at least one task is running yet no task can advance because they are all waiting on other tasks. In particular, we do not consider it a deadlock when a small group of tasks continue to communicate with each other but not the rest of the system, which remains blocked.

For each automaton, we generate an additional state bit (*changed*) that tracks whether it will proceed in this cycle or is blocked waiting for communication. Using additional state bits is unnecessary; in our first attempt we performed the check combinationally (i.e., in each state checked whether there was at least one task that could advance). However, introducing additional state bits improved the running time, so we adopted that style.

The rules we use for setting the *changed* bit for each automaton are similar to those for the automaton. The predicates are exactly the same, but instead of setting the state, the values set the *changed* bit to 1.

Once we have an *changed* bit for each automaton, the test for the absence of deadlock is simple: either at least one task was able to advance or the topmost task has terminated. This is easy to express in temporal logic for NuSMV:
AG(*root* != *exit* -> *changed_t1* | *changed_t2* | ...)
where *root* is the state of the topmost task and *advanced_tx* indicates that task *x* was able to make progress. In words, this says that in each state if the topmost task has not terminated then at least one task was able to make progress.

We also check whether a program will inevitably deadlock: if all possible paths lead to a deadlock state irrespective of the conditional predicates, then the program absolutely will deadlock. The temporal logic expression for its absence in NuSMV is
EG(*root* != *exit* -> *changed_t1* | *changed_t2* | ...)
I.e., in each state, if the topmost task is running, there is some path where at least one task was able to advance.

7.5 Results

We ran our deadlock detector on various SHIM programs on a 3 GHz Pentium 4 machine with 1 GB RAM. Table 7.1 lists our results. The Lines column shows for each example the number of lines of code including comments. The Channels and Tasks columns list the number of channels and concurrently running tasks we find after expanding the tasks into a tree and removing nontrivial tasks. Runtimes include the time taken for compilation, abstraction, generating the NuSMV model, and running the NuSMV model checker. We check for both the possibility and inevitability of a deadlock. As expected, the model checking time dominates on the larger examples. The Memory column reports the total resident set size used by the verifier. The States column reports the number of reachable states NuSMV found.

Source-Sink is a simple example centered around a pair of processes that pass data through a channel and print the results through an output channel. The large number of tasks arise from I/O functions used to print the output of this test case. Most of the examples here include many extra tasks for this reason.

Pipeline and the Prime Number Sieve are examples from Edwards and Zeng [52] that use bounded recursion. As mentioned earlier, we use their technique to remove the recursion before running NuSMV. The Sieve has many states because most of its tasks perform data-dependent communication and our model ends up considering all apparent possibilities, even though the system enters far fewer states in practice. Nevertheless, this illustrates the power of symbolic simulation: analyzing these three billion states takes NuSMV less than two seconds.

The Berkeley example contains a pair of tasks that communicate packets through a channel using a data-based protocol. After ignoring data, however, the tasks behave like simple sources and sinks, making it easy to prove the absence of deadlock.

The FIR filter is a parallel five-tap filter with twenty-eight tasks and channels (the core consists of seventeen tasks). Each task's automaton consists of a single loop (the filter is actually a synchronous dataflow model [76]) so the analysis is fairly easy.

Bitonic Sort is one of our most parallel examples: it uses twenty-four comparison tasks to order eight integers. All the additional tasks are sources, sinks, and (repeated calls to I/O routines). The communication behavior of the tasks is straightforward, but the tool has many tasks to consider.

Framebuffer is a 640 × 480 video framebuffer driven by a line-drawing task. Its communication is complicated.

The JPEG decoder is one of our largest applications to date, and illustrates some of the challenges in coding SHIM to avoid deadlocks. Our tool reported the possibility of a deadlock on the initial version (which actually works correctly) because of the code in Figure 7.7.

This code attempts to run three IDCT processors in parallel on an array of *n* macroblocks. For all but the last iteration of the loop, the dispatcher communicates on channels *I1*, *I2*, and *I3*, then receives data from *O1*, *O2*, and *O3*. However, since *n* may not be a multiple of three, this code is careful not to overrun the array bounds and may only perform one or two IDCT operations in the last iteration.

While this program works (provided the predicates on the *if* statements are written properly), our tool does not understand, say, the second and fourth conditionals are correlated and reports a potential deadlock.

Figure 7.8 illustrates how to avoid this problem by duplicating code and factoring it differently. Although our tool still treats the conditional branches as nondeterministic, it does not perceive a deadlock because, e.g., the synchronizations on I3 and O3 remain matched.

Figure 7.7, however, will not report an unavoidable deadlock because it has a non-deadlocking path.

Overall, our tool is able to quickly check these programs (in seconds) while using a reasonable amount of memory. While larger programs will be harder to verify, our technique is clearly practical for modestly sized programs.

```
{
  // ...
  for (int j = 0 ; j < n ; j += 3) {
    next I1 = iblock[j];
    if (j+1 < n) {
      next I2 = iblock[j+1];
      if (j+2 < n)
        next I3 = iblock[j+2];
    }
    oblock[j] = next O1;
    if (j+1 < n) {
      oblock[j+1] = next O2;
      if (j+2 < n)
        oblock[j+2] = next O3;
    }
  }
 // ...
} par {
  for (;;)
    next O1 = IDCT(next I1);
} par {
  for (;;)
    next O2 = IDCT(next I2);
} par {
  for (;;)
    next O3 = IDCT(next I3);
}
```

Figure 7.7: Fragment of the JPEG Decoder that causes our tool to report a deadlock; it ignores the correlation among *if* statements

7.6 Conclusions

We presented a static deadlock detection technique for the SHIM concurrent language. SHIM programs behave identically regardless of scheduling policy because

```
for(int j = 0 ; j < n ; j += 3) {
  next I1 = iblock[j];
  if (j+2 < n) {
    next I2 = iblock[j+1];
    next I3 = iblock[j+2];
    oblock[j] = next O1;
    oblock[j+1] = next O2;
    oblock[j+2] = next O3;
  } else if (j+1 < n) {
    next I2 = iblock[j+1];
    oblock[j] = next O1;
    oblock[j+1] = next O2;
  } else {
    oblock[j] = next O1;
  }
}
```

Figure 7.8: An equivalent version of the first task in Figure 7.7 for which our tool does not report a deadlock

they are based on Kahn networks [70]. This allows us to check for deadlock on synchronous models of SHIM programs and know the result holds for asynchronous implementations.

We expand each SHIM program into a tree of tasks, abstract each task as an automaton that performs communication and invokes and waits for its children, then express these automata in a form suitable for the NuSMV symbolic model checker. This is a mechanical process.

We abstract away data-dependent decisions when building each task's automaton. This both greatly simplifies their structure and can lead to false positives: our technique can report a program will deadlock even though it cannot. However, we believe this is not a serious limitation because there is often an alternative way to code a particular protocol that makes it insensitive to data and more robust to small modifications, i.e., less likely to be buggy.

Experimentally, we find NuSMV is able to detect or prove the absence of deadlock in seconds for modestly sized examples. We believe this is fast enough to make deadlock checking a standard part of the compilation process (i.e., instead of something too costly to run more than occasionally), which we believe is a first for concurrent languages.

We currently ignore exceptions in SHIM, which is safe but as a result, we may report as erroneous programs that throw exceptions to avoid a deadlock situation. While we do not know of any such programs, we plan to consider this issue in the future.

Although we were able to analyze programs efficiently, we further improved our technique by compositionally building the state space of SHIM programs. We discuss this algorithm in detail in the next chapter.

Chapter 8

Compositional Deadlock Detection for SHIM

Our previous chapter used NuSMV, a symbolic model checker, to detect deadlock in a SHIM program, but it did not scale well with the size of the problem. In this chapter, we take an incremental, divide-and-conquer approach to deadlock detection. We present a compositional deadlock detection technique for SHIM, in which tasks run asynchronously and communicate using synchronous CSP-style rendezvous. Although SHIM guarantees the absence of data races, a SHIM program may still deadlock if the communication protocol is violated.

SHIM's scheduling independence makes other properties easier to check because they do not have to be tested across all schedules; one is enough. Deadlock is one such property: for a particular input, a program will either always or never deadlock; scheduling choices cannot cause or prevent a deadlock. We exploited this property in the previous chapter, where we transformed asynchronous SHIM models into synchronous state machines and used the symbolic model checker NuSMV [34] to verify the absence of deadlock. This is unlike traditional techniques, such as Holzmann's SPIN model checker [63], in which all possible interleavings must be considered. While our technique was fairly effective because it could ignore interleavings, we improve upon it here.

In this chapter, we use explicit model checking with a form of assume-guarantee reasoning [100] to quickly detect the possibility of a deadlock in a SHIM program. Step by step, we build up a complete model of the program by forming the product machine of an automaton we are accumulating with another process from the program, each time checking the accumulated model for deadlock.

Our key trick: we simplify the accumulated automaton after each step, which often avoids exponential state growth. Specifically, we abstract away internal channels—those that do not appear in any other processes.

Figure 8.1 shows our technique in action. Starting from the (contrived) program, we first abstract the behavior of the first two tasks into simple automata. The first task communicates on channel a, then on channel b, then repeats; the second task does the same on channels b and c. We compose these automata by allowing either to take a step on unshared channels but insisting on a rendezvous when a channel is shared. Then, since channel b is local to these two tasks, we abstract away its behavior by merging two states. This produces a simplified automaton that we then compose with the automaton for the third task. This time, channel c is local, so again we simplify the automaton and compose it with the automaton for the fourth task.

The automaton we obtained for the first three tasks insists on communicating first on a then d; the fourth tasks communicates on d then a. This is a deadlock, which manifests itself as a state with no outgoing arcs.

For programs that follow such a pipeline pattern, the number of states grows exponentially with the number of pipeline stages (precisely, n stages produce 2^n states), yet our analysis only builds machines with $2n$ states before simplifying them to $n+1$ states at each step. Although we still have to step through and analyze each of the n stages (leading to quadratic complexity), this is still a substantial improvement.

Of course, our technique cannot always reduce an exponential state space to a polynomial one, but we find it often did on the example programs we tried.

In the rest of this chapter, we show how we check SHIM programs for deadlocks (Section 8.2) following our compose-and-abstract procedure described above. We present experimental results in Section 8.3 that shows our technique is superior to our earlier work using a symbolic model checking, review related work in Section 8.4, and conclude in Section 8.5.

8.1 An Example

Consider the SHIM program in Figure 8.2. The *main* function starts three tasks that communicate through channels a, b and c. The first task has a conditional statement, which we model as a nondeterministic choice. One of its branches synchronizes on channel a. The other branch synchronizes on both a and b. The second task synchronizes on channels a and c; the third task synchronizes on channels a and c, and then on b. The ownership is as follows: channel a is shared by all three tasks, channel b is shared by task 1 and task 3, and channel c is shared by tasks 2 and 3. This program does not deadlock. First all three tasks synchronize on channel a exhibiting multiway rendezvous. Next, tasks 2 and 3 rendezvous on

```
void main()
{
  chan int a, b, c, d;
  for(;;) {
    recv a; b = a + 1; send b;
  } par for(;;) {
    recv b; c = b + 1; send c;
  } par for(;;) {
    recv c; d = c + 1; send d;
  } par for(;;) {
    recv d; a = d + 1; send a;
  }
}
```

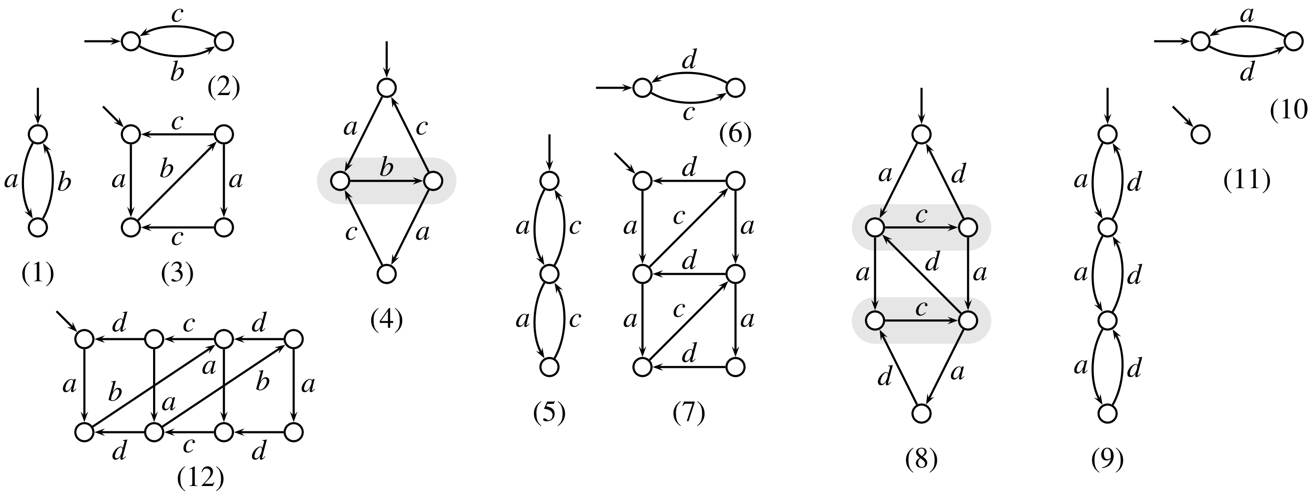


Figure 8.1: Analyzing a four-task SHIM program. Composing the automata for the first (1) and second (2) tasks gives a product automaton (3). Channel *b* only appears in the first two tasks, so we abstract away its effect by identifying (4) and merging (5) equivalent states. Next, we compose this simplified automaton (5) with that for the third task (6) to produce another (7). Now, channel *c* will not appear again, so again we identify (8) and merge (9) states. Finally, we compose this (9) with the automaton for the fourth task (10) to produce a single, deadlocked state (11) because the fourth task insists on communicating first on *d* but the other three communicate first on *a*. The direct composition of the first three tasks without removing channels (12) is larger—eight states.

```
void main()
{
  int i;
  chan int a, b;

  { // Task 1
    if (i % 10) {
      a = 1;
      send a;
    } else {
      a = 0;
      send a;
      recv b;
    }

  } par { // Task 2
    recv a;
    c = 1;
    send c;

  } par { // Task 3
    recv a;
    recv c;
    b = 10;
    send b;
  }
}
```

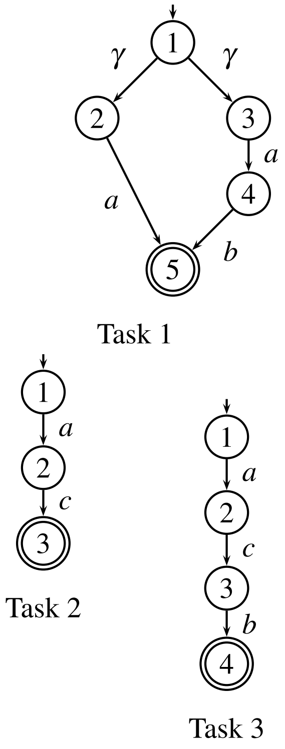


Figure 8.2: A SHIM program and the automata for its tasks

channel c. Task 3 then synchronizes with task 1 on channel *b* if the branch is not taken. Otherwise, it waits for task 1 to terminate and then does a dummy send on channel *b*. This is because task 3 is no longer compelled to wait for a terminated process (task 1).

We make the same assumptions as in the previous chapter. We assume the overall SHIM program is not recursive. We remove statically bounded recursion using Edwards and Zeng [52] and do not attempt to analyze programs with unbounded recursion.

Next, we duplicate code to force each function to have a unique call site. While this has the potential for an exponential increase in code size, we did not observe it.

We remove trivial functions—those that do not attempt to synchronize. A function is trivial if it does not attempt to send or receive and all its children (the spawned tasks) are trivial. Provided they terminate (an assumption we make), the behavior of such functions does not affect whether a SHIM program deadlocks. Fortunately, it appears that functions called in many places rarely contain communication (I/O functions are an exception), so the potential explosion from copying functions to ensure each has a unique call site rarely occurs in practice.

This preprocessing turns the call structure of the program into a tree, allowing us to statically enumerate all the tasks, the channels and their connections, and identify a unique parent and call site for each task (aside from the root).

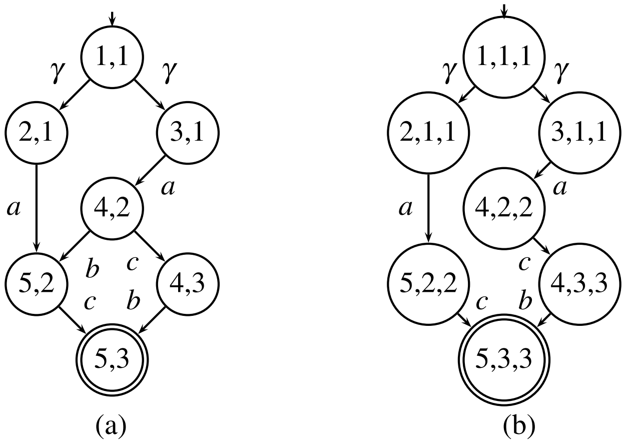


Figure 8.3: Composing (a) the automata for tasks 1 and 2 from Figure 8.2 and (b) composing this with task 3.

After preprocessing, we build a SHIM automaton for each task from the compiler's intermediate representation. A SHIM automaton has two kinds of arcs: channel and γ. A transition labeled with a channel name represents communication on that channel; a γ transition models conditionals (nondeterministic choices).

Figure 8.2 shows the three SHIM automata we construct for the program. The *if-else* in task 1 is modeled as state 1 with two outgoing γ transitions. On the other hand, we use arcs labeled by channels to represent communication.

Figure 8.3(a) shows the composition of tasks 1 and 2 from Figure 8.2. First, we compose task 1's state 1 with task 2's state 1. We create the (1,1) state with two outgoing γ transitions, and we then compose each of state 1's successor in task 1 with state 1 of task 2, generating states (2,1) and (3,1). At state (2,1), we can say that task 1 is at state 2 and task 2 is at state 1. We then add a transition from (2,1) to (5,2) labeled a because both tasks are ready to communicate on a in state (2,1). Similarly, we create state (4,2).

Then, at state (4,2), task 1 can fire b (in the absence of task 3) and task 2 can fire c. Since task 1 shares channel b but not c and task 2 shares channel c but not b, either transition is possible so we have two scheduling choices at state (4,2), which is represented by two transitions b and c from (4,2). By similar rules, we compose other states and finally we end up with Figure 8.3(a) as the result. The composed automaton owns channels a, b, and c.

Following the same procedure, we compose the automaton in Figure 8.3(a) with task 3 in Figure 8.2 to produce the automaton in Figure 8.3(b). We compose states in a similar fashion. However, when composing state (4,2) of Figure 8.3(a) with state 2 of task 3 in Figure 8.2, state (4,2)'s transition on channel b is not enabled because task 3 does not have a transition on b from its state 2. On the other hand, state (4,2)'s transition on channel c does not conflict with task 3, allowing us to transit from state (4,2,2) to state (4,3,3) on channel c in Figure 8.3(b).

8.2 Compositional Deadlock Detection

8.2.1 Notation

Below, we formalize our abstraction of SHIM programs. We wanted something like a finite automaton that could model the external behavior of a SHIM process (i.e., communication patterns).

We found we had to distinguish two types of choices: a nondeterministic choice induced by concurrency that can be made by the scheduler (i.e., selecting one of many enabled tasks) and control-flow choices made by the tasks themselves. Although a running task is deterministic (it makes decisions based purely on its state, which can be supplied in part by the [deterministic] series of data that arrive on its channels), we chose to abstract data computations to simplify the verification problem at the expense of rejecting some programs that would avoid deadlock in practice. Thus, we treat choices made by a task (e.g., at an if-else construct) as nondeterministic.

These two types of nondeterministic choices must be handled differently when looking for deadlocks: while it is acceptable for an environment to restrict choices that arise from concurrency, an environment cannot restrict choices made by the tasks themselves.

Our solution is an automaton with two types of edges: those labeled with channels representing communication, which need not all be followed when composing automata; and those labeled with γ, which we use to represent an internal choice made by a task and must be preserved when composing automata.

Definition 1 *A SHIM automaton a 6-tuple* $(Q,\Sigma,\gamma,\delta,q,f)$ *where* Q *is the set of states,* Σ *is the set of channels,* $\gamma \notin \Sigma$*,* $q \in Q$ *is the initial state,* $f \in Q$ *is the final state, and* $\delta = Q \times (\Sigma \cup \{\gamma\}) \to 2^Q$ *the transition function, where* $|\delta(s,c)| = 0$ *or* 1 *for* $c \neq \gamma$*.*

The δ transition function is key. For each state $s \in Q$ and channel $c \in \Sigma$, either $\delta(s,c) = \emptyset$ and the automaton is not ready to rendezvous on channel c in state s, or $\delta(s,c)$ is a singleton set consisting of the unique next state to which the automaton will transition if the environment rendezvous on c.

The special "channel" γ denotes computation internal to the system. If $\delta(s,\gamma) \neq \emptyset$, the automaton may transition to any of the states in $\delta(s,\gamma)$ from state s with no rendezvous requirement on the environment.

A state $s \in Q$ such that $\delta(s,c) = \emptyset$ for all $c \in \Sigma \cup \{\gamma\}$ corresponds to the system terminating normally if $s = f$ and is a deadlock state otherwise.

Next, we define how to run two SHIM automata in parallel. The main thing is that we require both automata to rendezvous on any shared channel.

Definition 2 *The composition* $T_1 \cdot T_2$ *of two SHIM automata* $T_1 = (Q_1,\Sigma_1,\gamma,\delta_1,q_1,f_1)$ *and* $T_2 = (Q_2,\Sigma_2,\gamma,\delta_2,q_2,f_2)$ *is* $(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \gamma, \delta, \langle q_1,q_2\rangle, \langle f_1,f_2\rangle)$*, where*

$$\delta(\langle p_1,p_2\rangle,\gamma) = \big(\delta_1(p_1,\gamma) \times \{p_2\}\big) \cup \big(\{p_1\} \times \delta_2(p_2,\gamma)\big),$$

and for $c \in \Sigma_1 \cup \Sigma_2$*,*

$$\delta(\langle p_1,p_2\rangle,c) = \begin{cases} \delta_1(p_1,c) \times \delta_2(p_2,c) & \text{when } c \in \Sigma_1 \cap \Sigma_2\text{;} \\ \delta_1(p_1,c) \times \{p_2\} & \text{when } c \in \Sigma_1 - \Sigma_2 \text{ or} \\ & p_2 = f_2\text{; and} \\ \{p_1\} \times \delta_2(p_2,c) & \text{when } c \in \Sigma_2 - \Sigma_1 \text{ or} \\ & p_1 = f_1 \end{cases}$$

Table 8.1: Comparison between compositional analysis (CA) and NuSMV

Program	Lines	Channels	Tasks	Deadlock?	Runtime (s)		Memory (MB)	
					CA	NuSMV	CA	NuSMV
Source Sink	35	2	11	No	0.004	0.004	1.31	6.28
Berkeley	49	3	11	No	0.01	0.01	2.6	5.96
Bitonic Sort	74	56	24	No	1.83	4.01	7.82	53.20
31-tap FIR Filter	218	150	120	No	0.2	21.10	21.06	63.33
Pipeline (1000 pipes)	1003	1000	1000	Yes	397.8	607.8	24.7	813
FFT (50 FFT tasks)	978	100	52	No	34.73	327	16.7	719
Frame Buffer	220	11	12	No	1.81	4.90	5.50	7.5
JPEG Decoder (30 IDCT processes)	2120	180	31	No	51.9	1177	16.06	203.44

Here, we defined two cases for the composed transition function. On γ (corresponding to an internal choice), either the first automaton or the second may take any of its γ transitions independently, hence the set union. Note that $\emptyset \times \{p_2\} = \emptyset$.

For normal channels, there are two cases. For a shared channel ($c \in \Sigma_1 \cap \Sigma_2$), both automata proceed simultaneously if each has a transition on that channel, i.e., have rendezvoused. For non-shared channels or if one of the tasks has terminated, the automaton connected to the channel may take a step independently (and implicitly assumes the environment is willing to rendezvous on the channel).

There should be no difference between running T_1 in parallel with T_2 and running T_2 in parallel with T_1, yet this is not obvious from the above definition. Below, we formalize this intuition by defining what it means for two automata to be equivalent, then showing the composition operator produces equivalent automata.

Definition 3 *Two SHIM automata* $T_1 = (Q_1, \Sigma_1, \gamma, \delta_1, q_1, f_1)$ *and* $T_2 = (Q_2, \Sigma_2, \gamma, \delta_2, q_2, f_2)$ *are* equivalent *(written* $T_1 \equiv T_2$*) if and only if* $\Sigma_1 = \Sigma_2$ *and there exists a bijective function* $b : Q_1 \to Q_2$ *such that* $q_2 = b(q_1)$, $f_2 = b(f_1)$, *and for every* $s \in Q_1$ *and* $c \in \Sigma_1 \cup \{\gamma\}$, $\delta_2(b(s), c) = b(\delta_1(s, c))$.

Lemma 1 *Composition is commutative:* $T_1 \cdot T_2 \equiv T_2 \cdot T_1$.

PROOF Follows from Definition 2 and Definition 3 by choosing $b(\langle p_1, p_2 \rangle) = \langle p_2, p_1 \rangle$. □

Lemma 2 *Composition is associative:* $(T_1 \cdot T_2) \cdot T_3 \equiv T_1 \cdot (T_2 \cdot T_3)$.

PROOF Follows from Definition 2, Lemma 1, and Definition 3 by choosing $b\big(\langle\langle p_1, p_2\rangle, p_3\rangle\big) = \langle p_1, \langle p_2, p3\rangle\rangle$. □

Algorithm 1 compose(automata list L)

```
1:  $T_1, \ldots, T_n$ = reorder($L$)
2:  $T = T_1$
3:  for i = 2 to n do
4:      $T = T \cdot T_i$ {Compose using Definition 2}
5:      $q =$ initial state of $T$
6:      for all channels $c$ in $T$ that are not in $T_{i+1}, \ldots, T_n$ do
7:          for all $\delta(p,c) = \{q\}$ do
8:              Set $\delta(p,c)$ to $\emptyset$ {p is the parent of q}
9:              Add $q$ to $\delta(p,\varepsilon)$
10:             Add $p$ to $\delta(q,\varepsilon)$
11:         end for
12:     end for
13:     $T =$ subset-construction($T$) {Remove $\varepsilon$ transitions}
14: end for
15: if $T$ has a deadlock state then
16:     return deadlock
17: else
18:     return no-deadlock
19: end if
```

8.2.2 Our Algorithm

We are finally in a position to describe our algorithm for compositional deadlock detection. Algorithm 1 takes a list of SHIM automata as input and returns either a composed SHIM automaton or failure when there is a deadlock. Since the order in which the tasks are composed does affect which automata are built along the way and hence memory requirements and runtime (but not the final result), the reorder function (called in Line 1) orders the automata based on a heuristic that groups tasks with identical channels. Once we compose tasks, we abstract away channels that are not used by other tasks, simplifying the composed automaton at each step.

We then compose tasks one by one. At each step we check if the composed automaton is deadlock free. We remove (Line 6 through Line 13) any channels that are local to the composed tasks (i.e., not connected to any other tasks). For every channel c, we find all the transitions on that channel (i.e., $\delta(p,c) = \{q\}$) and add ε transitions between states p and q. Then, we use the standard subset construction algorithm [3] to merge such states.

We do not abstract away channels connected to other tasks because the other tasks may impose constraints on the communication on these channels that lead to a deadlock. In general, adding another connected task often imposes order. For example, when task 1 and task 2 are composed, communications on *b* and *c* may occur in either order. This manifests itself as the scheduling choice at state (4,2) in Figure 8.3(a). However when task 3 is added, the communication on *c* occurs first.

The automata we produce along the way often have more behavior (interleavings) than the real system because at each point we have implicitly assumed that the environment will be responsive to whatever the automaton does. However, we never omit behavior, making our technique safe (i.e., we never miss a possible deadlock). Extra behavior generally goes away as we consider more tasks (our abstraction of data means that our automata are always over approximations, however). For example, when we compose Figure 8.3(a) with task 3, we get Figure 8.3(b). We get rid of the impossible case where communication on *b* appears before *c* generated in Figure 8.3(a).

We can only guarantee the absence of deadlock. Since we are ignoring data, we check for all branches in a conditional for deadlock freedom; even if one path fails, at best we can only report the possibility of a deadlock. It may be that the program does not in fact deadlock due to correlations among its conditionals.

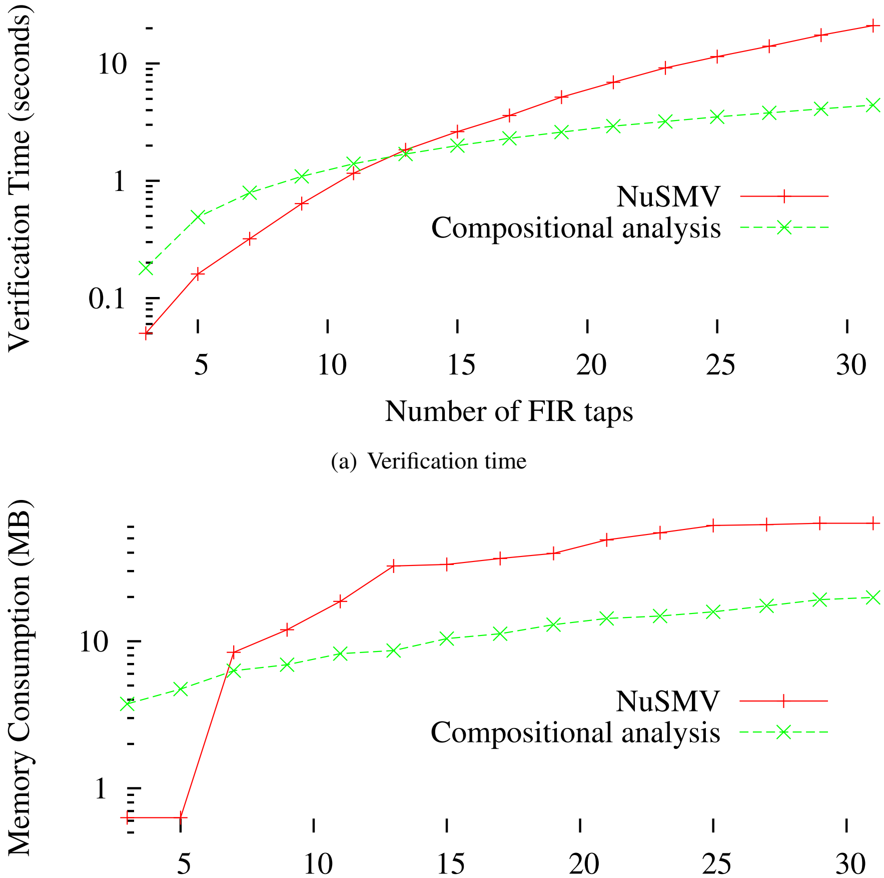
8.3 Experimental Results

We ran our compositional deadlock detector on the programs listed in Table 8.1 using a 3.2 GHz Pentium 4 machine with 1 GB memory. The *Lines* column lists the number of lines in the program; *Channels* is the number of channels declared in the program; *Tasks* is the number of non-trivial tasks after transforming the callgraph into a tree. *Deadlock?* indicates the outcome.

The *Runtime* columns list the number of seconds taken by both our new compositional tool and our previous work, which relies on NuSMV to model check the automaton. Similarly, the *Memory* columns compare the peak memory consumption of each.

We have used similar examples in the previous chapter. Source-Sink is a simple example centered around a pair of tasks that pass data through a channel and print the results through an output channel. The Berkeley example contains a pair of tasks that communicate packets through a channel using a data-based protocol. After ignoring data, the tasks behave like simple sources and sinks, so it is easy to prove the absence of deadlock. The verification time and memory consumption are trivial for both tools in these examples because they have simple communication patterns.

The Bitonic Sort example uses twenty-four comparison tasks that communicate on fifty-six channels to order eight integers. Although bitonic sort has twenty-four tasks, every channel is owned at most by 2 tasks, which gives our tool an opportunity to abstract away channels when it is not used by the rest of the tasks during composition. This helps to reduce the size of the automaton.

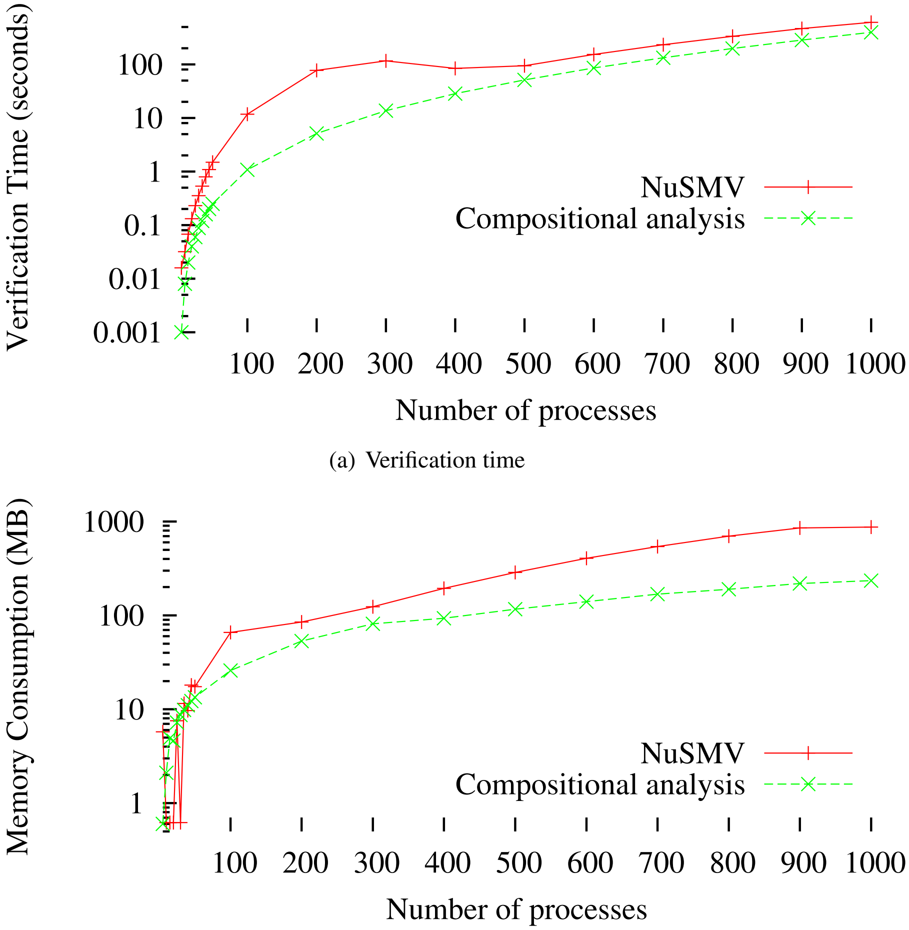


(a) Verification time

(b) Memory Consumption

Figure 8.4: n-tap FIR

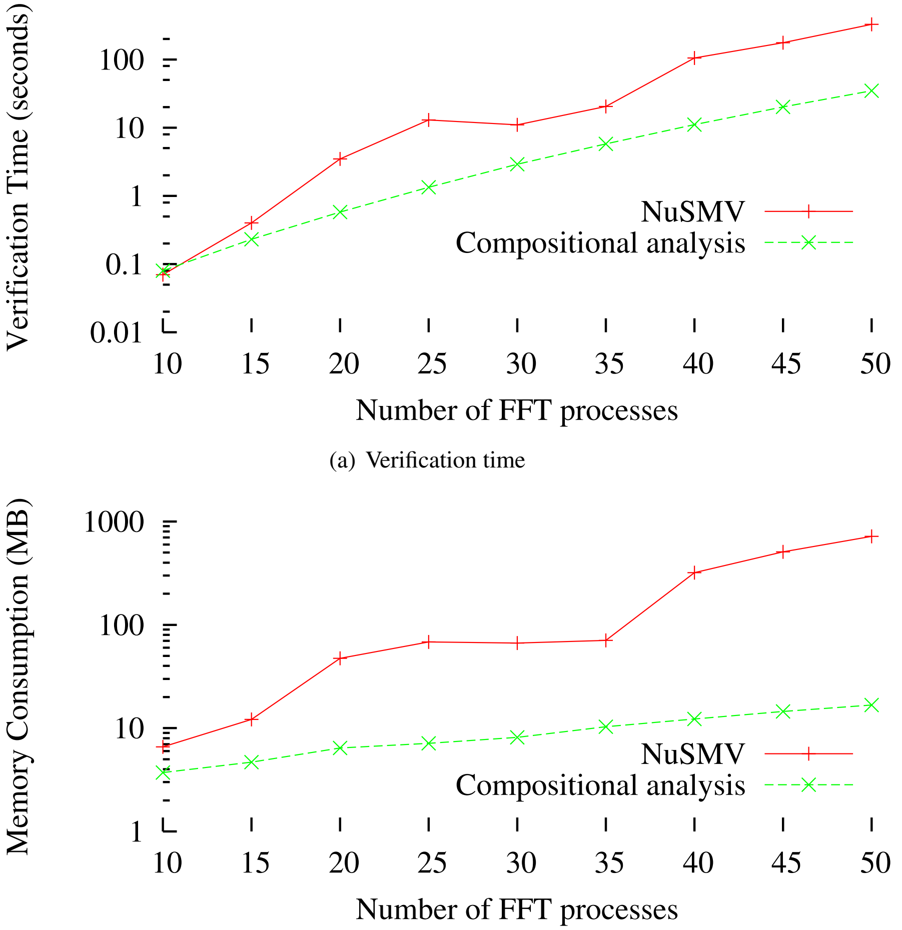
The FIR filter is a parallel 31-tap filter with 120 tasks and 150 channels. Each task consists of a single loop. Figure 8.4 compares our approach and NuSMV model checker for filters of sizes ranging from 3 to 31. The time taken by our tool grows quartically with the number of taps and exponentially with NuSMV. Figure 8.4(b) shows the memory consumption.



(a) Verification time

(b) Memory Consumption

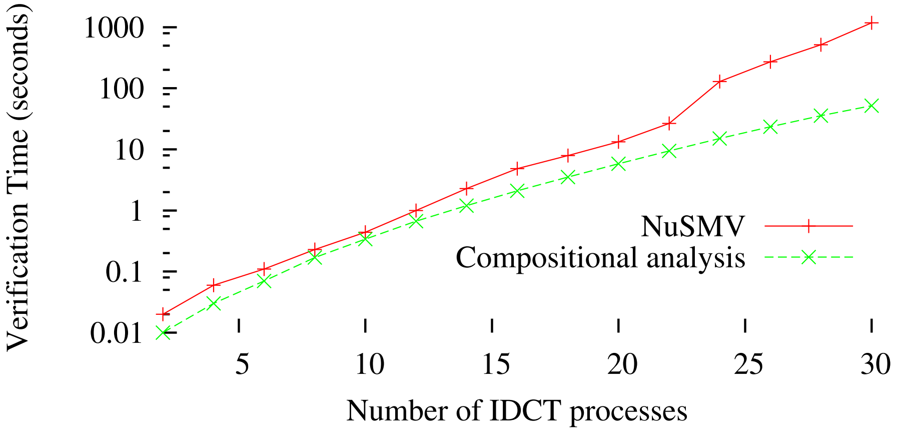
Figure 8.5: Pipeline



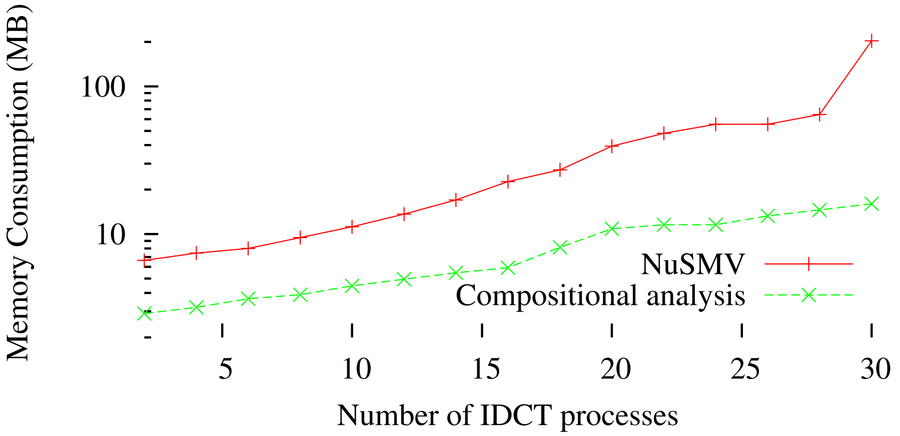
(a) Verification time

(b) Memory Consumption

Figure 8.6: Fast Fourier Transform



(a) Verification time



(b) Memory Consumption

Figure 8.7: JPEG Decoder

"Pipeline" is the example from Figure 8.1. Like the FIR, we tested our tool on a varying number of tasks. Although both tools seem to achieve $O(n^4)$ asymptotic time behavior, ours remains faster and uses less memory. Figure 8.8 illustrates how our tool performs exponentially on this example if we omit the channel abstraction step.

The FFT example is similar to the pipeline: most of the tasks' SHIM automata consist of a single loop. However, there is a master task that divides and communicates data to its slaves. The slaves and the master run in parallel. The master then waits for the processed data from each of the slaves. Figure 8.6 shows we perform much better as the size of the FFT increases.

The Framebuffer and JPEG examples are the only programs we tested with conditionals. Framebuffer is a 640×480 video framebuffer driven by a line-drawing task. It has a complicated, nondeterministic communication pattern, but is fairly small and not parametric. Our technique is still superior, but not by a wide margin.

The JPEG decoder is one of the largest applications we have written and is parametric. JPEG decoder has a number of parallel tasks, among which is a producer task that nondeterministically communicates with rest of the IDCT tasks. Figure 8.7(a) shows our tool exhibiting better asymptotic behavior than NuSMV.

Although our tool worked well on the examples we tried, it has some limitations. Our tool is sensitive to the order in which it composes automata. Although we use a heuristic to order the automata, it hardly guarantees optimality.

By design, our tool is not a general-purpose model checker; it cannot verify any properties other than the absence of deadlock. Furthermore, it can only provide abstract counter-examples because we remove channels during composition. We have not examined how best to present this information to a user.

Our compositional approach is forced to build the entire system for certain program structures. Consider the call graph shown in Figure 8.9. The main function forks two parallel tasks, f and g. Both f and g share channels $a_1, \ldots, a_n$. We first compose the children of f and then inline the composed children in f before composing f with g. If f is a pipeline program with a structure similar to the one described in Figure 8.1, when we compose f's children, we cannot abstract away any channel because g also owns all the channels. This leads to exponential behavior, but we find SHIM programs are not written like this in practice.

8.4 Related work

Many have tried to detect deadlocks in models with rendezvous communication. For example, Corbett [40] proposes static methods for detecting deadlocks in Ada

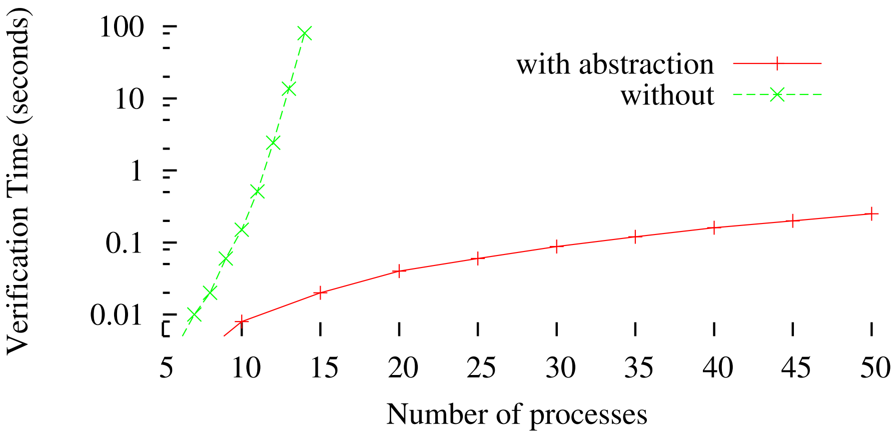


Figure 8.8: The importance of abstracting internal channels: verification times for the *n*-task pipeline with vs. without.

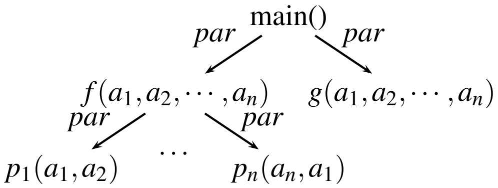


Figure 8.9: A SHIM program's call graph

programs. He uses partial order reduction, symbolic model checking, and inequality necessary conditions to combat state space explosion. However, these techniques do build the entire state space with some optimizations. These may be necessary for Ada, which does not have SHIM's scheduling independence. By contrast, we avoid building the complete state space by abstracting the system along the way. Masticola et al. [81] also propose a way to find deadlocks in Ada programs. Their technique is less precise because they use approximation analysis that runs in polynomial time. Secondly, their method only applies to a subset of Ada programs. By contrast, our technique can be applied to any SHIM program, but can run in exponential time on some.

Compositional verification is a common approach for alleviating the state explosion problem. It decomposes a system into several components, verifies each component separately, and infers the system's correctness. This approach verifies the properties of a component in the absence of the whole system. Two variants of the method have been developed: assume-guarantee [100] and compositional minimization [35].

In the assume-guarantee paradigm, assumptions are first made about a component, then the component's properties are verified under these assumptions. However, it is difficult to automatically generate reasonable assumptions, often requiring human intervention. Although there has been significant work on this [10; 26; 39; 60; 90], Cobleigh et al. [38] report that, on average, assume-guarantee reasoning does not show significant advantage over monolithic verification either in speed or in scalability. Compared to assume-guarantee reasoning, which verifies a system top down with assumptions, our work incrementally verifies the system bottom up. In addition, the assumptions we make along the way are somehow trivial: the environment is assumed to be merely responsive to our tasks' desire to rendezvous.

Instead of assuming an environment, compositional minimization models the environment of a component using another component called the interface and reasons about the whole system's correctness through inference rules. Krimm et al. [99] implemented this algorithm to generate state space from Lotos programs, then extended their work [73] to detect deadlocks in CSP programs with partial order reduction. Our work is similar in that we iteratively compose an interface with a component and later simplify the new interface by removing channels and merging equivalent states. However, they provide little experimental evidence about how their algorithm scales or compares with traditional model checkers.

Zheng et al. [135] apply the compositional minimization paradigm to hardware verification. They propose a failure-preserving interface abstraction for asynchronous design verification. To reduce complexity, they use a fully automated interface refinement approach before composition. Our channel abstraction technique is analogous to their interface refinement, but we apply it to asynchronous software instead of synchronous hardware.

There are many other compositional techniques for formal analysis. Berezin et al. [14] survey several compositional model checking techniques used in practice and discuss their merits. For example, Chaki et al. [24; 25] and Bensalem et al. [12] combine compositional verification with abstraction-refinement methodology. In other words, they iteratively abstract, compose and refine the system's components, once a counter example is obtained. By contrast, we do not apply any refinement techniques but build the system incrementally to even find a counter example.

Compared to the previous chapter on deadlock detection in SHIM, what we present here uses explicit model checking, incremental model building, and on-the-fly abstraction instead of throwing a large model at a state-of-the art symbolic model checker (we used NuSMV [34]). Experimentally, we find the approach we present here is better for all but the smallest examples.

8.5 Conclusions

We presented a static deadlock detection technique for the SHIM concurrent language. The semantics of SHIM allow us to check for deadlock in programs compositionally without loss of precision.

We expand a SHIM program into a tree of tasks, abstract each as a communicating automaton, then compose the tasks incrementally, abstracting away local channels after each step.

We have compared our compositional technique with the previous chapter (which used the NuSMV general-purpose model checker) on different examples with varying problem sizes. Experimentally, we find our compositional algorithm is able to detect or prove the absence of deadlock faster: on the order of seconds for large-sized examples. We believe this is fast enough to make deadlock checking a regular part of the compilation process.

In both the methods, we abstract away data-dependent decisions when building each task's automaton. This both greatly simplifies their structure and can lead to false positives: our technique can report a program will deadlock even though it cannot. However, we believe this is not a serious limitation because there is often an alternative way to code a particular protocol that makes it insensitive to data and more robust to small modifications. A more robust, but less efficient solution is to implement a runtime deadlock detection algorithm which we discuss in the next chapter.

Chapter 9

Runtime Deadlock Detection for SHIM

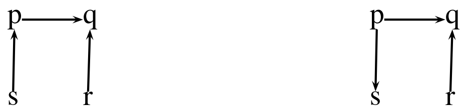
Our deadlock detection techniques for SHIM in the previous chapters may give false positives, because they operate at compile time and abstract away data. To avoid this problem, we designed a runtime deadlock detection technique. If the static technique reports that a program is deadlock free, the program is indeed deadlock free. However, if it reports a program erroneous, the program may actually not deadlock. In the latter case, we would like to switch on the runtime deadlock detection technique; i.e., use the runtime deadlock detection technique (which adds extra overhead) only when necessary.

In this chapter, we design a runtime technique for detecting deadlocks in SHIM. We also provide a mechanism for deterministically breaking deadlocks and resuming execution in SHIM programs. We discuss the algorithm in detail in the next section.

9.1 The Algorithm

SHIM is deterministic but not deadlock free. However, the deadlocks are reproducible [122]; a deadlock that occurs with one schedule will always occur under another schedule for a given input.

To remove deadlocks, we maintain a dependency graph during runtime. The vertices of the graph are task numbers. SHIM's runtime system runs the deadlock detection algorithm. Whenever a task p calls *send* on a channel, it waits for a peer task q to do its counterpart operation *recv* on the same channel. If task q is also ready to communicate, then the two tasks rendezvous and the communication is successful. On the other hand if task q is not ready and doing some other work, then



(a) A possible SHIM network

(b) An **impossible** SHIM network. *p* has two outgoing edges

Figure 9.1: Possible and impossible configurations of tasks in the SHIM model

task *p* indicates that it is waiting, by adding an edge from *p* to *q* in the dependency graph. Then, *p* checks if there is a path from *q* leading back to itself. If there is a cycle, then the program has a deadlock. This cycle-finding algorithm is not expensive because of the following reason. By SHIM semantics, at any instant, a task can at most block on one channel. Therefore, there is at most one outgoing edge from any task *p*. See Figure 9.1. Consequently, our cycle finding algorithm takes time linear in the number of tasks.

Since every task updates edges originating from its vertex in the shared-dependency graph, the addition of edges by two tasks to the shared dependency graph can be done concurrently. This is because no two tasks are going to add the same edge (i.e., an edge with the same end vertices).

Two or more tasks can check for a cycle concurrently and at least one task in the deadlock will detect a cycle. This is because every task adds the edge first and then checks for a cycle. If a cycle is found, the deadlock-breaking mechanism is initiated. The first task to detect a cycle, clears the cycle by removing the edges in the dependency graph and signals all other blocked processes in the cycle to revive.

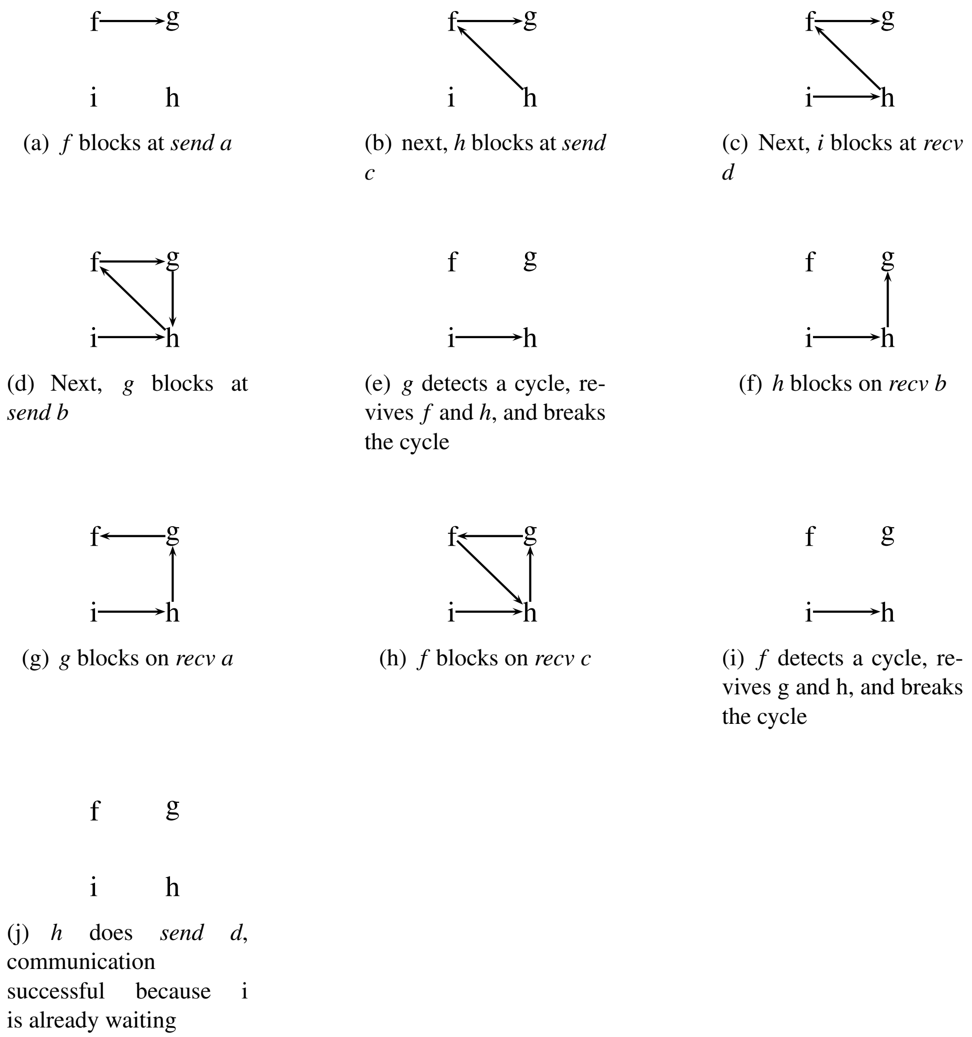
All revived tasks (including the task that signalled) now complete their communication by not waiting for their counterpart operations. A revived *recv* operation receives the last value seen on the channel. A revived *send* value puts the new value on the channel by performing a dummy write.

We will now run our technique on a simple example shown in Figure 9.2. There are four tasks running simultaneously. Task *f*'s *send a* waits for *g*'s *recv a*. Task *g*'s *send b* waits for *h*'s *recv b*. Task *h*'s *send c* waits for *f*'s *recv c*. In the absence of a deadlock breaker, the three tasks wait for each other causing a deadlock.

If we break the deadlocks in the program, the program will terminate. The deadlock-breaking technique for Figure 9.2 is shown in Figure 9.3. Suppose *f* calls *send a* first (Figure 9.3(a)), it realizes that *g* is not ready to receive *a* and therefore *f* adds an edge from vertex *f* (itself) to vertex *g* in the dependency graph.

```
void f(chan int &a, chan int c)
{
   send a = 1; /* Deadlocking action */
   /* Writes 1 to channel ‘a’ after the deadlock is broken */
   recv c; /* Another deadlocking action */
   /* Receives 3 after the deadlock is broken */
}

void g(chan int &b, chan int a)
{
   send b = 2; /* Deadlocking action */
   /* Writes 2 to channel ‘b’ after the deadlock is broken */
   recv a; /* Another deadlocking action */
   /* Receives 1 after the deadlock is broken */
}

void h (chan int &c, chan int b, chan int &d)
{
   send c = 3; /* Deadlocking action */
   /* Writes 3 to channel ‘c’ after the deadlock is broken */
   recv b; /* Another deadlocking action */
   /* Receives 2 after the deadlock is broken */
   send d = 4;
}

void i (chan int d)
{
   recv d; /* Receives 4 */
}

main() {
   /* Create 4 channels and initializes them with 0 */
   chan int a = 0, b = 0, c = 0, d = 0;
   /* Run f, g, h and i in parallel */
   f(a, c) par g(b, a) par h(c, b, d) par i(d);
   /* Here: a = 1, b = 2, c = 3, d = 4 */
}
```

Figure 9.2: Another example of a deadlock and the effect of our deadlock-breaking algorithm



(a) *f* blocks at *send a*

(b) next, *h* blocks at *send c*

(c) Next, *i* blocks at *recv d*

(d) Next, *g* blocks at *send b*

(e) *g* detects a cycle, revives *f* and *h*, and breaks the cycle

(f) *h* blocks on *recv b*

(g) *g* blocks on *recv a*

(h) *f* blocks on *recv c*

(i) *f* detects a cycle, revives g and h, and breaks the cycle

(j) *h* does *send d*, communication successful because i is already waiting

Figure 9.3: Building the Dependency Graph for Figure 9.2

It then checks if there is a cycle. Since the cycle has not yet formed, f suspends itself. Next, if h calls *send c* (Figure 9.3(b)), it finds that f is not ready to receive c and therefore h adds an edge from vertex h (itself) to vertex f, sees that there is no cycle and suspends itself. Next, if i calls *recv d* (Figure 9.3(c)), it realizes that h is not yet ready to *send d*. Therefore i adds an edge from vertex i to vertex h, sees that there is no cycle and suspends itself. Next, g calls *send b* (Figure 9.3(d)) and it adds an edge from vertex g (itself) to h.

After g adds an edge from itself to h, g runs the deadlock detection algorithm and detects a cycle. It (Figure 9.3(e)) now removes the edges in the cycle, revives all the tasks in the deadlock. Now the revived tasks go ahead with their operations (without waiting for the counterparts). f writes 1 to channel a, g writes 2 to channel b and h writes 3 to channel c This modifies *main*'s copy of a, b and c. The three tasks then move to their next statements.

Next, the tasks f, g and h deadlock again on their *recv*'s forming a cycle (Figures 9.3(f), 9.3(g) and 9.3(h)). The deadlock is broken by one of the tasks (task f in Figure 9.3(i)). f receives whatever was last put on the channel c which is 3. Similarly, g receives 1, h receives 2. Then tasks f and g terminate. Now task h calls *send d* (Figure 9.3(j)) and finds that i is ready to receive on channel d. The two tasks h and i rendezvous to communicate, and they finally terminate.

The advantages of our method are that the deadlock detection technique can run concurrently and in linear time. However, two or more tasks may detect a cycle simultaneously; therefore we need only one of the tasks to take the responsibility of reviving other tasks. We therefore require some sort of synchronization to break the deadlock but not to detect a deadlock.

9.2 Conclusions

The runtime deadlock detector is the contribution of this chapter. Before any deadlock, the execution of the SHIM program will be deterministic because of the property of the SHIM model. The deadlock-breaking step is deterministic because it just advances all the deadlocked tasks. The program is deterministic after the deadlock is broken, because the remaining statements are executed normally following SHIM's principle. Therefore, we still maintain determinism even after introducing a runtime deadlock breaker to the basic SHIM model.

Deadlock detection algorithms cost time on general graphs. SHIM's constraint of never waiting on two channels sidesteps this problem, rendering the cycle-finding algorithm linear time.

There are a number of runtime distributed deadlock detecting algorithms. Chandy, Misra, and Haas [28] is among the best known. According to their technique, whenever a process, say *i*, is waiting on a process, say *j*, *i* sends a probe message to *j*. *j* sends the same message to all the processes it is waiting on and so on. If the probe message comes back to *i*, then *i* reports a deadlock.

Like others, Chandy et al. concentrate on the multiple-path problem where multiple edges may leave a single vertex. Probe messages must be duplicated at these nodes. We can apply the same algorithm to our setting, but since we have at most one outgoing edge per vertex, we do not have to duplicate messages.

In summary, our technique efficiently addresses the two major pitfalls of concurrent programming – nondeterminism and deadlocks. SHIM provides determinism. The static and runtime deadlock-detection techniques provide deadlock freedom. We do not claim that our methods are the best; we believe that this chapter and the preceding ones will provide insight on achieving both determinism and deadlock freedom, and the ideas here can be used while programmers think of designing concurrent models and languages.

Chapter 10

D^2C: A Deterministic Deadlock-free Concurrent Programming Model

The SHIM programming model is interesting but does not guarantee deadlock freedom by semantics. We need explicit deadlock detection techniques to catch deadlocks in programs. In this chapter, we provide an extension of SHIM — a concurrent model that is both deterministic and deadlock free. Any program that uses this model is guaranteed to produce the same output for a given input. Additionally, the program will never deadlock: the program will either terminate or run for ever.

10.1 Approach

Nondeterminism arises primarily due to read-write and write-write conflicts. In the D^2C model, we allow multiple tasks to write to a shared variable concurrently, but we define a commutative, associative reduction operator that will operate on these writes.

The program in Figure 10.1 creates three tasks in parallel f, g and h. f and g are modifying x. Even though f and g are modifying x concurrently, f sees the effect of g only when it executes *next*. Similarly g sees the effect of f only when it executes *next*. When a task executes *next*, it waits for all tasks that share variables with it, to also execute *next*. The *next* statement is like a barrier. At this statement, the shared variables are reduced using the reduction operator. In the example in Figure 10.1, the reduction operator is $+$ because x is declared with a reduction operator $+$ in line 24. Therefore after the *next* statement, the value of x

```
void f(shared int &a) {
 /* a is 0 */
  a = 3;
 /* a is 3 , x is still 0 */
  next;  /* The reduction operator is applied */
 /* a is now 8, x is 8 */
}

void g(shared int &b) {
 /* b is 0 */
  b = 5;
 /* b is 5, x is still 0 */
  next;  /* The reduction operator is applied */
 /* b is now 8, x is 8 */
}

void h (shared int &c) {
 /* c is 0 , x is still 0 */
  next;
 /* c is now 8, x is 8 */
}

void main() {
  shared int (+) x = 0;
  /* If there are multiple writers, reduce
     using the + reduction operator */
  f(x); par g(x); par h(x);
  /* x is 8 */
}
```

Figure 10.1: A D^2C program

is 3 + 5 which is 8 and it is reflected everywhere. Function *h* also rendezvous with *f* and *g* by executing *next* and thus it obtains the new value 8.

The *next* synchronization statement is deadlock free. We do not give a formal proof here, but it follows from the fact that the *next* statement is a conjunctive barrier on all shared variables. On contrast, SHIM is not deadlock free. Also, they do not allow multiple tasks to write to a shared variable because they provide ownership to variables.

10.2 Implementation

We implemented our model in the X10 programming language [29]. As described in Chapter 12, X10 is a parallel, distributed object-oriented language. To a Java-like sequential core it adds constructs for concurrency and distribution through the concepts of activities and places. An activity is a unit of work, like a thread in Java; a place is a logical entity that contains both activities and data objects. X10 uses the Cilk model of task parallelism and a task scheduler similar to that of Cilk.

Our preliminary implementation is as follows. We did a very conservative analysis to check if a particular shared variable is being used by multiple tasks. If yes, we force the variable to be shared with a reduction operator. This forces race freedom. Otherwise, the compiler throws an error.

Each thread maintains a copy of the shared variable. A thread always reads from or writes to its local copy. Whenever the *next* statement is called, all threads sharing the variable synchronize. The last thread to synchronize does a linear reduction of the local copies using the commutative, associative operator in the variable declaration. It then updates the local copies with the new value.

10.3 Results

To test the performance of our model, we ran a number of benchmarks on a 1.6 GHz Quad-Core Intel Xeon (E5310) server running Linux kernel 2.6.20 with SMP (Fedora Core 6). The processor "chip" actually consists of two dice, each containing a pair of processor cores. Each core has a 32 KB L1 instruction and a 32 KB L1 data cache, and each die has a 4 MB of shared L2 cache shared between the two cores. Figure 10.2 shows the results. We measured the deterministic implementation of the applications with the original implementation. A bar with value below 1 indicates that the deterministic version ran slower than the original version.

The AllReduce Example is a parallel tree based implementation of reduction. The Pipeline example passes data through a number of intermediate stages; at each stage the data is processed and passed on to the next stage. Convolve is an application of the Pipeline program.

The N-Queens Problem finds the number of ways in which N queens can be placed on an N*N chessboard such that none of them attack each other. The MontiPi application finds the value of π using Monte-Carlo simulation. The K-Means program partitions n data points into k clusters concurrently.

The Histogram program sorts an array into buckets based on the elements of the array. The Merge Sort program sorts an array of integers. The Prefix example operates on an array and the resulting array is obtained from the sum of the elements in the original array up to its index.

The SOR, IDEA, RayTrace, LUFact, SparseMatMul and Series programs are JGF benchmarks. The RayTrace r benchmarks renders an image of sixty spheres. It has data dependent array access.

The SOR example performs Jacobi successive relaxation on a grid; it continu-

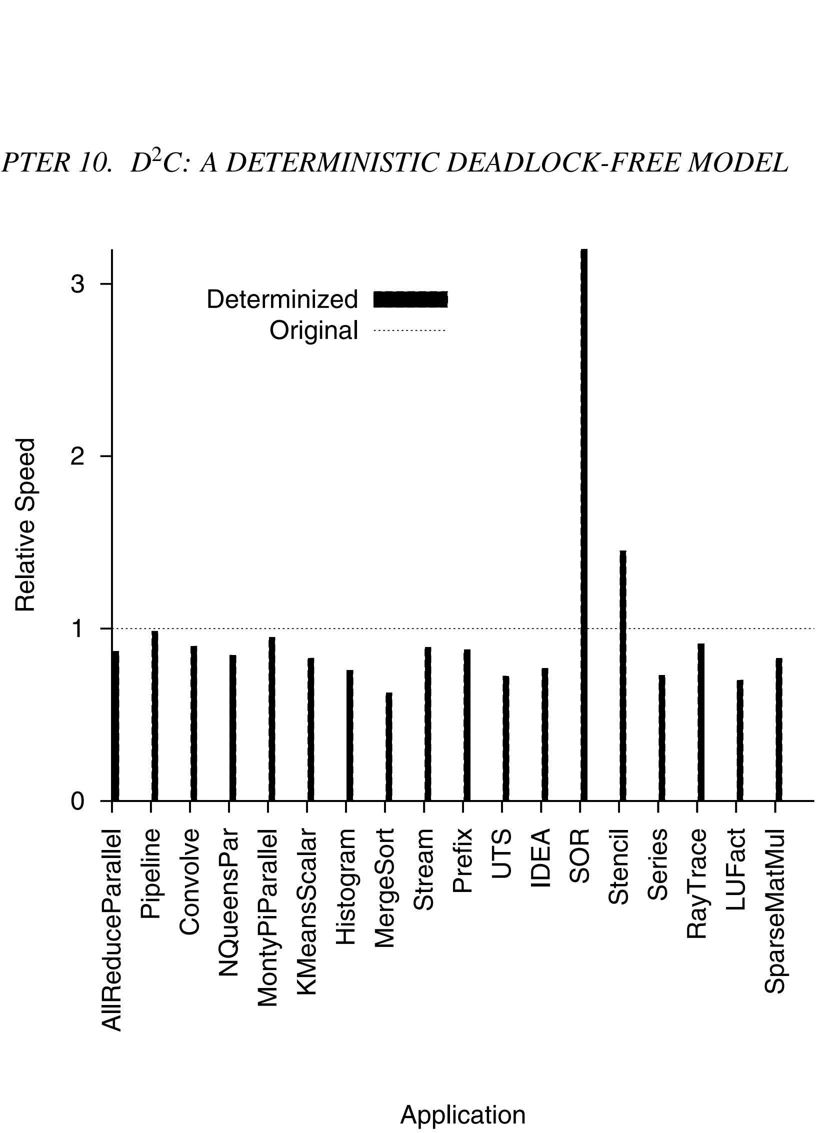


Figure 10.2: Experimental Results

ously updates a location of the grid based on the location's neighbors. The Stencil program is the 1-D version of the SOR.

The LUFact application transforms an N*N matrix into upper triangular form. The Series benchmark computes the first N coefficients of the function $f(x) = (x+1)^x$. The IDEA benchmark performs International Data Encryption algorithm (IDEA) encryption and decryption on an array of bytes. The SparseMatMul program performs multiplication of two sparse matrices.

The UTS benchmark [92] performing an exhaustive search on an unbalanced tree. It counts the number of nodes in the implicitly constructed tree that is parameterized in shape, depth, size, and imbalance.

For most of the examples, the deterministic version had a performance degradation of 1% - 25% as expected. However, for some examples like SOR and Stencil, the deterministic version performed better. The original version of these examples had explicit 2-phased barriers to differentiate between reads and writes, while the deterministic version requires just a single phase, because we maintain a local copy in each thread to eliminate read-write conflicts. Hence, the deterministic version performed better.

10.4 Conclusions

We have presented a deterministic, deadlock free model. We have a proof (not shown here) that formulates this hypothesis. We have added these features as constructs to the X10 programming language. We also plan implement it as a library. A number of examples fit into this model: Histogram, Convolution, UTS, Sparse Matrix Multiplication etc.

As future work, we plan to allow user defined reduction operators in our language. We therefore require a mechanism to check for associativity and commutativity of these operators. Secondly, we would like to use static analysis to improve the runtime efficiency of these constructs. Thirdly, we would like to implement this as a library, and check the program to see if it does not override the deterministic library. Next, we would like to build a determinizing tool [125] like Kendo [93] and [42] based on D^2C.

The D^2C model is advantageous that it does not introduce deadlocks by semantics. However, the *next* statement, being conjunctive, enforces more centralized synchronization than SHIM. This may be disadvantageous in terms of efficiency for some programs, but we do not see this kind of behavior in our benchmarks.

Our ultimate goal is efficient concurrency with determinism and deadlock freedom. D^2C will introduce a way of bug-free parallel programming that will enable programmers to shift easily from sequential to parallel worlds and this will be a necessary step along the way to pervasive parallelism in programming.

Part IV

Improving Efficiency

Outline

Part II provides efficient techniques to generate code from SHIM programs. Our goal is to reduce synchronization overhead as much as possible. In this part, we further improve the efficiency of SHIM constructs and related deterministic constructs in other languages. We show both compile-time and runtime techniques to optimize these constructs and their lower level implementations.

Chapter 11

Reducing Memory in SHIM programs

In this chapter, we present a static analysis technique for improving memory efficiency in SHIM programs. We focus on reducing memory consumption by sharing buffers among tasks, which use them to communicate using CSP-style rendezvous. We determine pairs of buffers that can never be in use simultaneously and use a shared region of memory for each pair.

We need to optimize space because embedded systems generally have limited memory. Overlays, which amount to time multiplexing the use of memory regions, is one way to reduce a program's memory consumption. In this chapter, we propose a technique that automatically finds opportunities to safely overlay communication buffer memory in SHIM.

Our technique produces a static abstraction of a SHIM program's dynamic behavior, which we then analyze to find buffers that can share memory. Experimentally, we find our technique runs quickly on modest-sized programs and can sometimes reduce memory requirements by half.

SHIM processes communicate through channels. The sequence of symbols transmitted over each channel is deterministic but the relative order of symbols between channels is generally undefined. If the sequences of symbols transmitted over two channels do not interfere, we can safely share buffers. Our technique establishes ordering between pairs of channels. If we cannot find such an ordering, we conclude that the pair cannot share memory.

Our analysis is conservative: if we establish two channels can share buffers, they can do so safely, but we may miss opportunities to share certain buffers because we do not model data and may treat the program as separate pieces to avoid an exponential explosion in analysis cost. Specifically, we build sound abstractions

```
void main()
{
 chan int a, b, c;
 {

   // Task 1
   next a = 6; // Send a (synchronize with task 2)

 } par {

   // Task 2
   next a; // Receive a (synchronize with task 1)
   next b = a + 1; // Send 7 on b (synchronize with task 3)

 } par {

   // Task 3
   next b; // Receive b (synchronize with task 2)
   next c = b + 1; // Send 8 on c (synchronize with task 4)

 } par {

   // Task 4
   next c; // Receive c (synchronize with task 3}
   // c = 8 here
 }
}
```

Figure 11.1: A SHIM program that illustrates the need for buffer sharing

to avoid state space explosions, effectively enumerating all possible schedules with a product machine.

One application of our technique is to minimize buffer memory used by code generated by the SHIM compiler for the Cell Broadband engine in Chapter 5. The heterogeneous Cell processor [69] consists of a power processor element (PPE) and eight synergistic processor elements (SPEs). The SHIM compiler maps tasks onto each of the SPEs. Each SPE has its own local memory and shares data through the PPE. The PPE synchronizes communication and holds all the channel buffers in its local memory. The SPE communicates with the PPE using mailboxes [72].

We wish to reduce memory used by the PPE by overlapping buffers of different channels. Our static analyzer does live range analysis on the communication channels and determines pairs of buffers that are never live at the same time. We demonstrate in Section 11.5 that the PPE's memory usage can be reduced drastically for practical examples such as a JPEG decoder and an FFT.

In this chapter, we address an optimizing technique for SHIM: buffer sharing.

In the program in Figure 11.1, the main task starts four tasks in parallel. Tasks 1 and 2 communicate on *a*. Then, tasks 2 and 3 communicate on *b* and finally tasks 3 and 4 on *c*. The value of *c* received by task 4 is 8. Communication on *a* cannot occur simultaneously with that of *b* because task 2 forces them to occur sequentially them. Similarly communications on *b* and *c* are forced to be sequential by task 3. Communications on *a* and *c* cannot occur together because they are forced to be sequential by the communication on *b*. Our tool understands this pattern and reports that *a*, *b*, and *c* can share buffers because their communications never overlap, thereby reducing the total buffer requirements by 66% for this program.

Below, we model a SHIM program's behavior to analyze buffer usage (Section 11.1), and describe how we compose models of SHIM tasks to build a product machine for the whole program (Section 11.2), how we avoid state explosion (Section 11.3), and how we use these results to reduce buffer memory usage (Section 11.4). We present experimental results in Section 11.5 and related work in Section 11.6.

11.1 Abstracting SHIM Programs

First, we assume that a SHIM program has no recursion. We use the techniques of Edwards and Zeng [52] to remove bounded recursion, which makes the program finite and renders the buffer minimization problem decidable. We do not attempt to analyze programs with unbounded recursion.

Although the recursion-free subset of SHIM is finite state and therefore tractable in theory, in practice the full state space of even a small program is usually too large to analyze exactly; a sound abstraction is necessary. A SHIM task has both computation and communication, but because buffers are used only when tasks communicate, we abstract away the computation.

Since we abstract away computation, we must assume that all branches of any conditional statement can be taken. This leaves open the possibility that two channels may appear to be used simultaneously but in fact never are, but we believe our abstraction is reasonable. In particular it is safe: we overlap buffers only when we are sure that two channels can never be used at the same time regardless of the details of the computation.

11.1.1 An Example

In Figure 11.2, the *main* function consists of two tasks that communicate through channels *a*, *b*, and *c*.

The first task communicates on channels *a* and *b* in a loop; the second task

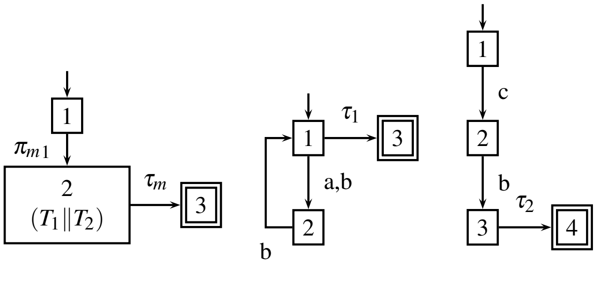
```
void main() {
  chan int a, b, c;
  {

    // Task 1
    for (int i = 0; i < 15; i++) { // state 1
      if (i % 2 == 0)
        next a = 5;
      else
        next b = 7;
      // state 2
      next b = 10;
    }
    // state 3

  } par {

    // Task 2
    // state 1
    next c = 13;
    // state 2
    next b;
    // states 3 & 4
  }
}
```

Figure 11.2: A SHIM program

synchronizes on channels c and b, then terminates. Once a task terminates, it is no longer compelled to synchronize on the channels to which it is connected. Thus after the second task terminates, the first task just talks to itself; i.e., it is the only process that participates in a rendezvous on its channels. Terminated processes do not cause other processes to deadlock.

At compilation time, the compiler dismantles the main function of Figure 11.2 into tasks T_1 and T_2. T_1 is connected to channels a and b since a and b appear in the code section of T_1. Similarly T_2 is connected to channels b and c. During the first iteration of the loop in T_1, T_1 talks to itself on a; since no other task is connected to a. Meanwhile, T_2 talks to itself on c. Then the two tasks rendezvous on b, communicating the value 10, then T_2 terminates. During subsequent iterations of T_1, T_1 talks to itself on either b twice or a and b once each.

In the program in Figure 11.2, communication on b cannot occur simultaneously with that on c because T_2 forces the two communications to be sequential and therefore b and c can share buffers. On the other hand, there is no ordering between channels a and c; a and c can rendezvous at the same time and therefore



(a) M (a,b,c,π_{m1}) (b) T_1 (a,b,τ_1) (c) T_2 (b,c,τ_2)

Figure 11.3: The main task and its subtasks

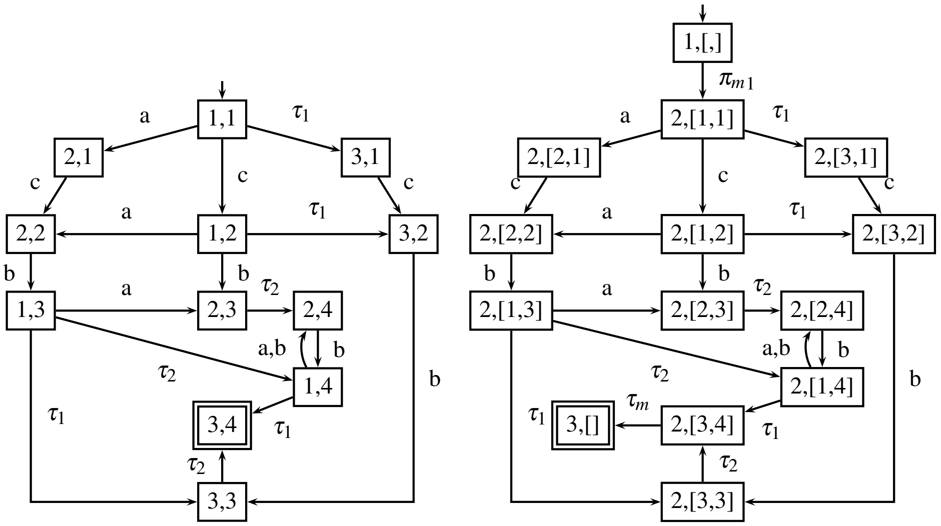
a and *c* cannot share buffers. By overlapping the buffers of *b* and *c*, we can save 33% of the total buffer space.

Our analysis performs the same preprocessing as our static deadlock detector in Chapter 7. It begins by removing bounded recursion using Edwards and Zeng's technique [52]. Next, we duplicate functions to force every call site to be unique. This has the potential of producing an exponential blow up, but we have not observed it in practice.

At this point, the call graph of the program is a tree, enabling us to statically determine all the tasks and the channels to which each is connected.

Next we disregard all functions that do not affect the communication behavior of the program. Because we are ignoring data, their behavior cannot affect whether we consider a buffer to be sharable. We implicitly assume every such function can terminate—again, a safe approximation.

Next, we create an automaton that models the control and communication behavior for each function. Figure 11.3 shows automata for the three tasks (main, T_1, and T_2) of Figure 11.2. For each task, we build a deterministic finite state automaton whose edges represent choices, typically to communicate. The states are labeled by program counter values and the transitions by channel names. Each automaton has a unique final state, which we draw as a double box. There is a transition from every terminating state to this final state labeled with a dummy channel that indicates such a transition. An automaton has only one final state but can have multiple terminating states. In the T_1 of Figure 11.2, state 1 is the terminating state, state 3 is the final state, and they are connected by τ_1, which is like a classical ε transition. However, a true ε transition would make the automaton nondeterministic, so we instead create the dummy channel τ_1 that is unique to T_1 and allow T_1 to freely move from state 1 to state 3 without having to synchronize



(a) $T_1 \cdot T_2$ (a,b,c,τ_1,τ_2)

(b) $M \cdot T_1 \cdot T_2$(a,b,c,τ_1,τ_2,π_{m1})

Figure 11.4: Composing tasks in Figure 11.3: (a) Merging T_1 and T_2. (b) Inlining $T_1 \cdot T_2$ in M.

with any other another task.

The main function has a dummy π_{m1} transition from its start to the entry of state 2 ($T_1 \| T_2$), which represents the *par* statement in *main*. In general, we create a dummy channel for every *par* in the program.

Figure 11.4(a) shows the product of T_1 and T_2—an automaton that represents the combined behavior of T_1 and T_2. We constructed Figure 11.4(a) as follows. We start with state (program counter) values $(1,1)$. At this point, T_1 can communicate on a and move to state 2. Therefore we have an arc from $(1,1)$ to $(2,1)$ labeled a. Similarly, T_2 can communicate on c and move to its state 2. From state $(1,1)$ it is not possible to communicate on b because only T_1 is ready to communicate, not T_2 (T_2 is also connected to b). Also at state $(1,1)$, T_1 can terminate by taking the transition τ_1 and moving to $(3,1)$.

From state $(3,1)$, T_2 can transition first to state $(3,2)$ by communicating on channel c and then to state $(3,3)$ by communicating on b; these transitions do not change the state of T_1 because it has already terminated.

From $(2,1)$, T_2 can communicate on c and change the state to $(2,2)$. Similarly from $(1,2)$, T_1 can communicate on a and move to $(2,2)$. In state $(1,2)$ it is also possible to communicate on b since both tasks are ready. Therefore, we have an arc b from $(1,2)$ to $(2,3)$. Since T_1 may also choose to terminate in state $(1,2)$, there is an arc from $(1,2)$ to $(3,2)$ on τ_1. Other states follow similar rules.

To determine which channels may share buffers, we consider all states that have two or more outgoing edges. For example, in Figure 11.4(a), state $(1,1)$ has outgoing transitions on *a* and *c*. Either of them can fire, so this is a case where the program may choose to communicate on either *a* or *c*. This means the contents of both of these buffers are needed at this point, so we conclude buffers for *a* and *c* may not share memory. We prove this formally later in the chapter.

From Figure 11.2, it is evident that *a* and *b* can never occur together because T_1 forces them to be sequential. However, since state $(1,2)$ has outgoing transitions on *a* and *b*, our algorithm concludes that *a* and *b* can occur together. However, they actually can not. We draw this erroneous conclusion because our algorithm does not differentiate between scheduling choices and control flow choices (i.e., due to conditionals such as *if* and *while*). By doing this we are only adding extra behavior to the system and disregarding pairs of channels whose buffers actually could be shared. This is not a big disadvantage because our analysis remains safe. For this example, our algorithm only allows *b* and *c* to share buffers.

Figure 11.4(b) is obtained by inlining the automaton for $T_1 \cdot T_2$—Figure 11.4(a)—within M. This represents the entire program in Figure 11.2. Since the *par* call is blocking, inlining $T_1 \cdot T_2$ within M is safe. We replaced state 2 of Figure 11.3(a) with Figure 11.4(a) to obtain Figure 11.4(b). The conclusions are the same as that of Figure 11.4(a)—only *b* and *c* can share buffers.

11.2 Merging Tasks

In this subsection, we use notation from automata theory to formalize the merging of two tasks. We show our algorithm does not generate any false negatives and is therefore safe.

Definition 4 *A* deterministic finite automaton T *is a 5-tuple* $(Q, \Sigma, \delta, q, f)$ *where* Q *is the set of states,* Σ *is the set of channels,* $q \in Q_1$ *is the initial state,* $f \in Q$ *is the final state, and* $\delta \subseteq Q \times \Sigma \rightarrow Q$ *is the partial transition function.*

Definition 5 *If* T_1 *and* T_2 *are automata, then the* composed automaton $T_1 \cdot T_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, \langle q_1, q_2 \rangle, \langle f_1, f_2 \rangle)$*, where, for* $\langle p_1, p_2 \rangle \in Q_1 \times Q_2$ *and* $a \in$

$\Sigma_1 \cup \Sigma_2$,

$$\delta_{12}(\langle p_1, p_2\rangle, a) = \begin{cases} \langle \delta_1(p_1,a), & \textit{if } a \in \sum_1 \textit{ and } a \in \sum_2\textit{;} \\ \quad \delta_2(p_2,a)\rangle & \\ \langle \delta_1(p_1,a), p_2\rangle & \textit{if } a \in \sum_1 \textit{ and} \\ & (a \notin \sum_2 \textit{ or } p_2 = f_2\textit{);} \\ \langle p_1, \delta_2(p_2,a)\rangle & \textit{if } a \in \sum_2 \textit{ and} \\ & (a \notin \sum_1 \textit{ or } p_1 = f_1\textit{);} \\ \textit{undefined} & \textit{otherwise;} \end{cases}$$

is the transition rule for composition.

In general, if T_1 has m states and T_2 has n, then the product $T_1 \cdot T_2$ can have at most mn states. The states are labeled by a tuple composed of the program counter values of the individual tasks. Each state can have at most k outgoing edges, where k is the total number of channels. Consequently, the total number of edges in the graph can at most be mnk (k accounts for the extra τ and π channels—one extra channel per task and one per *par*).

Below, we demonstrate that the order in which automata are composed does not matter. Although the state labels will be different, the states are isomorphic. First, we define exactly what we mean for two automata to be equivalent.

Definition 6 *Two automata $T_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$ and $T_2 = (Q_2, \Sigma_2, \delta_2, q_2, f_2)$ are* equivalent
(written $T_1 \equiv T_2$) if and only if $\Sigma_1 = \Sigma_2$ and there exists a bijective function $b : Q_1 \to Q_2$ such that $q_2 = b(q_1)$, $f_2 = b(f_1)$, and for every $p \in Q_1$ and $a \in \Sigma_1$, either both $\delta_1(p,a)$ and $\delta_2(b(p),a)$ are defined and $\delta_2(b(p),a) = b(\delta_1(p,a))$ or both are undefined.

Lemma 3 *Composition is commutative: $T_1 \cdot T_2 \equiv T_2 \cdot T_1$.*

PROOF By definition,

$$\begin{aligned} T_1 \cdot T_2 &= (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, \langle q_1, q_2\rangle, \langle f_1, f_2\rangle) \text{ and} \\ T_2 \cdot T_1 &= (Q_2 \times Q_1, \Sigma_2 \cup \Sigma_1, \delta_{21}, \langle q_2, q_1\rangle, \langle f_2, f_1\rangle). \end{aligned}$$

We claim $b(\langle p_1, p_2\rangle) = \langle p_2, p_1\rangle$ is a suitable bijective function. First, note $\Sigma_1 \cup \Sigma_2 = \Sigma_2 \cup \Sigma_1$, $\langle q_2, q_1\rangle = b(\langle q_1, q_2\rangle)$, and $\langle f_2, f_1\rangle = b(\langle f_1, f_2\rangle)$.

Next,

$$\begin{aligned}
&\delta_{21}(b(\langle p_1, p_2\rangle), a)\\
&\quad = \delta_{21}(\langle p_2, p_1\rangle, a)\\
&\quad = \begin{cases} \langle \delta_2(p_2,a), \delta_1(p_1,a)\rangle & \text{if } a \in \sum_2 \text{ and } a \in \sum_1;\\ \langle \delta_2(p_2,a), p_1\rangle & \text{if } a \in \sum_2 \text{ and}\\ & (a \notin \sum_1 \text{ or } p_1 = f_1);\\ \langle p_2, \delta_1(p_1,a)\rangle & \text{if } a \in \sum_1 \text{ and}\\ & (a \notin \sum_2 \text{ or } p_2 = f_2);\\ \text{undefined} & \text{otherwise;} \end{cases}\\
&\quad = b\left(\begin{cases} \langle \delta_1(p_1,a), \delta_2(p_2,a)\rangle & \text{if } a \in \sum_1 \text{ and } a \in \sum_2;\\ \langle p_1, \delta_2(p_2,a)\rangle & \text{if } a \in \sum_2 \text{ and}\\ & (a \notin \sum_1 \text{ or } p_1 = f_1);\\ \langle \delta_1(p_1,a), p_2\rangle & \text{if } a \in \sum_1 \text{ and}\\ & (a \notin \sum_2 \text{ or } p_2 = f_2);\\ \text{undefined} & \text{otherwise;} \end{cases} \right)\\
&\quad = b\big(\delta_{12}(\langle p_1, p_2\rangle, a)\big)
\end{aligned}$$

Thus, $T_1 \cdot T_2 \equiv T_2 \cdot T_1$. □

Lemma 4 *Composition is associative:* $(T_1 \cdot T_2) \cdot T_3 \equiv T_1 \cdot (T_2 \cdot T_3)$.

PROOF By definition,

$$\begin{aligned}
(T_1 \cdot T_2) \cdot T_3 &= ((Q_1 \times Q_2) \times Q_3, (\Sigma_1 \cup \Sigma_2) \cup \Sigma_3, \delta_{(12)3},\\
&\qquad \langle\langle q_1, q_2\rangle, q_3\rangle, \langle\langle f_1, f_2, \rangle, f_3\rangle)\\
T_1 \cdot (T_2 \cdot T_3) &= (Q_1 \times (Q_2 \times Q_3), \Sigma_1 \cup (\Sigma_2 \cup \Sigma_3), \delta_{1(23)},\\
&\qquad \langle q_1, \langle q_2, q_3\rangle\rangle, \langle f_1, \langle f_2, f_3\rangle\rangle).
\end{aligned}$$

We claim $b(\langle\langle p_1, p_2\rangle, p_3\rangle) = \langle p_1, \langle p_2, p_3\rangle\rangle$ is a suitable bijective function. First, note that $(\Sigma_1 \cup \Sigma_2) \cup \Sigma_3 = \Sigma_1 \cup (\Sigma_2 \cup \Sigma_3)$, $\langle q_1, \langle q_2, q_3\rangle\rangle = b(\langle\langle q_1, q_2\rangle, q_3\rangle)$, and $\langle f_1, \langle f_2, f_3\rangle\rangle = b(\langle\langle f_1, f_2\rangle, f_3\rangle)$.

Next,

$$
\begin{aligned}
&\delta_{1(23)}(b(\langle\langle p_1,p_2\rangle,p_3\rangle),a)\\
&= \delta_{1(23)}(\langle p_1,\langle p_2,p_3\rangle\rangle,a)\\
&= \begin{cases}
\langle\delta_1(p_1,a),\langle\delta_2(p_2,a), & \text{if } a\in\sum_1 \text{ and } a\in\sum_2 \text{ and}\\
\qquad\qquad\delta_3(p_3,a)\rangle\rangle & a\in\sum_3;\\
\langle\delta_1(p_1,a),\langle\delta_2(p_2,a),p_3\rangle\rangle & \text{if } a\in\sum_1 \text{ and } a\in\sum_2 \text{ and}\\
 & (a\notin\sum_3 \text{ or } p_3=f_3);\\
\langle\delta_1(p_1,a),\langle p_2,\delta_3(p_3,a)\rangle\rangle & \text{if } a\in\sum_1 \text{ and } a\in\sum_3 \text{ and}\\
 & (a\notin\sum_2 \text{ or } p_2=f_2);\\
\langle\delta_1(p_1,a),\langle p_2,p_3\rangle\rangle & \text{if } a\in\sum_1 \text{ and}\\
 & (a\notin\sum_2 \text{ or } p_2=f_2) \text{ and}\\
 & (a\notin\sum_3 \text{ or } p_3=f_3);\\
\langle p_1,\langle\delta_2(p_2,a),\delta_3(p_3,a)\rangle\rangle & \text{if } a\in\sum_2 \text{ and } a\in\sum_3 \text{ and}\\
 & (a\notin\sum_1 \text{ or } p_1=f_1);\\
\langle p_1,\langle\delta_2(p_2,a),p_3\rangle\rangle & \text{if } a\in\sum_2 \text{ and}\\
 & (a\notin\sum_1 \text{ or } p_1=f_1) \text{ and}\\
 & (a\notin\sum_3 \text{ or } p_3=f_3);\\
\langle p_1,\langle p_2,\delta_3(p_3,a)\rangle\rangle & \text{if } a\in\sum_3 \text{ and}\\
 & (a\notin\sum_1 \text{ or } p_1=f_1) \text{ and}\\
 & (a\notin\sum_2 \text{ or } p_2=f_2);\\
\text{undefined} & \text{otherwise;}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&= b\left(\begin{cases}
\langle\langle\delta_1(p_1,a),\delta_2(p_2,a)\rangle, & \text{if } a\in\sum_1 \text{ and } a\in\sum_2 \text{ and}\\
\qquad\qquad\delta_3(p_3,a)\rangle & a\in\sum_3;\\
\langle\langle\delta_1(p_1,a),\delta_2(p_2,a)\rangle,p_3\rangle & \text{if } a\in\sum_1 \text{ and } a\in\sum_2 \text{ and}\\
 & (a\notin\sum_3 \text{ or } p_3=f_3);\\
\langle\langle\delta_1(p_1,a),p_2\rangle,\delta_3(p_3,a)\rangle & \text{if } a\in\sum_1 \text{ and } a\in\sum_3 \text{ and}\\
 & (a\notin\sum_2 \text{ or } p_2=f_2);\\
\langle\langle\delta_1(p_1,a),p_2\rangle,p_3\rangle & \text{if } a\in\sum_1 \text{ and}\\
 & (a\notin\sum_2 \text{ or } p_2=f_2) \text{ and}\\
 & (a\notin\sum_3 \text{ or } p_3=f_3);\\
\langle\langle p_1,\delta_2(p_2,a)\rangle,\delta_3(p_3,a)\rangle & \text{if } a\in\sum_2 \text{ and } a\in\sum_3 \text{ and}\\
 & (a\notin\sum_1 \text{ or } p_1=f_1);\\
\langle\langle p_1,\delta_2(p_2,a)\rangle,p_3\rangle & \text{if } a\in\sum_2 \text{ and}\\
 & (a\notin\sum_1 \text{ or } p_1=f_1) \text{ and}\\
 & (a\notin\sum_3 \text{ or } p_3=f_3);\\
\langle\langle p_1,p_2\rangle,\delta_3(p_3,a)\rangle & \text{if } a\in\sum_3 \text{ and}\\
 & (a\notin\sum_1 \text{ or } p_1=f_1) \text{ and}\\
 & (a\notin\sum_2 \text{ or } p_2=f_2);\\
\text{undefined} & \text{otherwise;}
\end{cases}\right)\\
&= b\big(\delta_{(12)3}(\langle\langle p_1,p_2\rangle,p_3\rangle,a)\big)
\end{aligned}
$$

Thus, $(T_1\cdot T_2)\cdot T_3 \equiv T_1\cdot(T_2\cdot T_3)$. □

Lemma 5 $T_1\cdot T_2\cdot T_3\cdots T_n \equiv (((T_1\cdot T_2)\cdot T_3)\cdots)\cdot T_n$

PROOF Since the composition is commutative and associative, we can build the entire system incrementally by composing two tasks at a time. □

Lemma 6 *The outgoing transitions from a given state represent every possible rendezvous that can occur at that particular state.*

PROOF According to the definition of δ, we add an outgoing edge to a state for every rendezvous that can happen immediately after that state.

Multiple outgoing arcs from a state may represent choices due to control statements (such as *if* or *while*). $\delta(p_1,a) = q_2$ and $\delta(p_1,b) = q_2$, then we have two outgoing choices due to control flow.

On the other hand, a scheduling choice may occur when composing two tasks. A scheduling choice occurs when the ordering between two rendezvous is unknown. This happens when two different pairs of tasks can rendezvous on two different channels at the same time.

Suppose $a \in \Sigma_1$ and $a \notin \Sigma_2$ and $\delta_1(p_1,a) = q_1$, and if $b \in \Sigma_2$ and $b \notin \Sigma_1$ and $\delta_2(p_2,b) = q_2$, then $\delta_{12}(\langle p_1, p_2\rangle, a) = \langle q_1, p_2\rangle$ and $\delta_{12}(\langle p_1, p_2\rangle, b) = \langle p_1, q_2\rangle$. Thus, for every possible scheduling choice, we have an outgoing edge from the given state.

The absence of any choice due to control or scheduling will leave it with either one or zero outgoing arcs. Consequently, the outgoing transitions from a given state represent all possible rendezvous that can occur at that particular state. They represent both control flow and scheduling choices. □

A scheduling choice imposes no ordering among rendezvous, thus allowing the possibility of two or more rendezvous to happen at the same time.

Theorem 1 *Two channels a and b can share buffers if,* $\forall p$, *at most one of* $\delta(p,a)$ *and* $\delta(p,b)$ *is defined, but not both.*

PROOF Suppose a and b can rendezvous at the same time and if p_1 represents the state of the program counter just before the rendezvous, then by Lemma 6 we have two outgoing arcs from p_1: $\delta(p_1,a) = q_1$ and $\delta(p_1,b) = q_2$

Consequently, for some p, both $\delta(p,a)$ and $\delta(p,b)$ exists. Conversely, if for all p at most one of $\delta(p,a)$ and $\delta(p,b)$ exist, then we can safely say that a and b can share buffers. □

Our algorithm does not differentiate between control flow choices (e.g., due to *if* or *while*) and scheduling choices (due to partial ordering of rendezvous). Both kinds of choices produce states having multiple outgoing arcs. We conclude that arcs going out from the same state cannot share buffers. The multiplicity can be contributed only by control choices leading to false positives, but our system is safe; whenever we are unsure, we do not allow sharing.

11.3 Tackling State Space Explosion

If two tasks communicate infrequently, there is a possibility that the number of states in the product machine will grow too large to deal with. We address this by introducing a threshold, which limits the stack depth of our recursive product machine composition procedure and corresponds to the longest simple path in the product machine. If we reach the threshold, we stop and treat the two tasks being composed as being separate (i.e., unable to share buffers between them).

This heuristic, which we chose because our implementation was running out of stack space on certain complex examples, has the advantage of applying exactly when we are unlikely to find opportunities to share buffer memory. Tightly coupled tasks tend to have small state spaces—these are exactly those that allow buffer memory to be shared. Loosely coupled tasks by definition run nearly independently and thus the communication pattern of most pairs of channels are uncontrolled, eliminating the chance to share buffers between them.

Algorithm 2 is the composition algorithm. It recursively composes two states p_1 and p_2. The *depth* variable is initialized to 0 and incremented whenever successor states are composed. Whenever *depth* exceeds the threshold, we declare failure.

Algorithm 2 compose(p_1, p_2, Σ_1, Σ_2, *depth*, *threshold*)

```
if $depth > threshold$ then
  print "Threshold exceeded"
else
  for all $a \in \Sigma_1 \cup \Sigma_2$ do
    $\langle q_1, q_2 \rangle = \delta(\langle p_1, p_2 \rangle, a)$
    if $\langle q_1, q_2 \rangle \notin hash$ then
      Add $\langle q_1, q_2 \rangle$ to $hash$
      compose($q_1$, $q_2$, $\Sigma_1$, $\Sigma_2$, $depth + 1$, $threshold$)
    end if
  end for
end if
```

We draw conclusions about local channels (whose scope has been completely explored) and we remain silent about the others. We make safe conclusions even when other channels have not been completely explored.

Theorem 2 *If our algorithm concludes that two channels a and b can share buffers after abstracting away channel c, then a and b can still share buffers in the presence of c.*

Example	Lines	Channels	Tasks	Bytes Saved	Buffer Reduction	Runtime	States
Source-Sink	35	2	11	4	50 %	0.1 s	394
Pipeline	35	5	9	16388	25	0.1	68
Bitonic Sort	35	5	13	12	60	0.1	135
Prime Number Sieve	40	5	16	12	60	0.5	122
Berkeley	40	3	11	4	33.33	0.6	285
FIR Filter	110	28	28	52	46.43	13.8	74646
Framebuffer	185	11	16	28	0.002	1.3	15761
FFT	230	14	15	344068	50	0.6	3750
JPEG Decoder	1020	7	15	772	50.13	1.8	517

Table 11.1: Experimental results with the threshold set to 8000

PROOF If *a* and *b* can share buffers, then there is a sequential ordering between them. By SHIM semantics [49], introduction of a new channel can create ordering between two channels that are not ordered, but can never disrupt an existing sequential ordering. Therefore, if our algorithm concludes that two buffers can share channels, introducing a new channel does not affect the conclusion. □

We conclude that two channels can share buffers only if two conditions hold: the two channels have been explored completely and every state has at most one of the two channels in its outgoing edge set.

We take a bottom-up approach while merging groups of tasks. Tasks in a (preprocessed) SHIM program have a tree structure that arises from nesting of *par* constructs. We merge the leaf tasks of this tree before merging their parents. We stop merging when all tasks have exceeded the threshold or if the complete program has been merged. This approach allows us to stop whenever we run out of time or space without violating safety.

11.4 Buffer Allocation

Our static analysis algorithm produces a set S that contains pairs of channels that can share buffers. Let S' be the complement of this set. We represent it as a graph: channels represent vertices and for every pair $\langle c_i, c_j \rangle \in S'$, we draw an edge between c_i and c_j. Two adjacent vertices cannot share buffers. Every node has a weight, which corresponds to the size of the channel.

Minimizing buffer memory consumption, therefore, reduces to the weighted vertex covering problem [80; 79]: a graph G is colored with p colors such that no two adjacent vertices are of the same color. We denote the maximum weight of a vertex colored with color i as $\max(i)$, and we need to find a coloring such that $\sum_{i=1}^{p} \max(i)$ is minimum. The problem is NP-hard.

We use a greedy first-fit algorithm to get an approximate solution. Let G be a

Threshold	Bytes Saved	Buffer Reduction	Runtime	States
2000	0	0 %	0.6 s	10024
3000	0	0	1.5	23530
4000	0	0	3.4	51086
5000	52	46.43	12.4	70929
6000	52	46.43	12.8	72101
7000	52	46.43	13.5	73433
8000	52	46.43	13.8	74646

Table 11.2: Effect of threshold on the FIR filter example

list of groups. Initially *G* is empty. We order the channels in nonincreasing order of buffer sizes, then add the channels one by one to the first nonconflicting group in *G*. If there is no such group, we create a new group in *G* and add the channel to this newly created group. A group is defined to be nonconflicting if the channel to be added can share its buffer with every channel already in the group. Channels in the same group can share buffers. This algorithm runs in polynomial time but does not guarantee an optimal solution.

11.5 Experimental Results

We implemented our algorithm and ran it on various SHIM programs. Table 11.1 lists the results of running the experiments on a 3 GHz Pentium 4 Linux machine with 1 GB RAM. For each example, the columns list the number of lines of code in the program, the total number of channels it uses, the number of tasks that take part in communication (i.e., excluding any functions that perform no communication), the number of bytes of buffer memory saved by applying our algorithm, what percentage this is of overall buffer memory, the time taken for analysis (including compilation, abstraction, verification, and grouping buffers), and the number of states our algorithm explored. For these experiments, we set the threshold to 8000. We use the same benchmarks from the previous chapters.

Specifically, it takes about thirteen seconds to analyze the FIR program and the number of states explored is about eighty thousand. Since this was one of the more challenging examples for our algorithm, we tried varying the threshold. Table 11.2 summarizes our results. As expected, the number of visited states increases as we increase the threshold. With a threshold of 1000, we hardly explore the program, but higher thresholds let us explore more. When the threshold reaches 5000, we have explored enough of the system to begin to find opportunities for sharing buffer memory, even though we have not explored the system completely.

Experimentally, we find that the analysis takes less than a minute for modestly large programs and that we can reduce buffer space by 60% and therefore considerable amount of PPE memory on the Cell processor for examples like the bitonic sort and the prime number sieve.

11.6 Related Work

Many memory reduction techniques exist for embedded systems. Greef et al. [41] reduce array storage in a sequential program by reusing memory. Their approach has two phases: they internally reduce storage for each array, then globally try to share arrays. By contrast, our approach looks for sharing opportunities globally on communication buffers in a concurrent setting.

StreamIt [120] is a deterministic language like SHIM. Sermulins et al. [108] present cache aware optimizations that exploit communication patterns in StreamIt programs. They aim to improve instruction and data locality at the cost of data buffer size. Instead, we try to reduce buffer sizes.

Chrobak et al. [33] schedule tasks in a multiprocessor environment to minimize maximum buffer size. Our algorithm does not add scheduling constraints to the problem: it reduces the total buffer size without affecting the schedule, and thereby not affecting the overall speed.

The techniques of Murthy et al. [86; 87; 88; 89], Teich et al. [118], and Geilen et al. [57] are closest to ours. They describe several algorithms for merging buffers in signal processing systems that use synchronous data flow models [76]. Govindarajan et al. [58] minimize buffer space while executing at the optimal computation rate in dataflow networks. They cast this as a linear programming problem. Sofronis et al. [112] propose an optimal buffer scheme with a synchronous task model as basis. These papers revolve around minimizing buffers in a synchronous setting; our work solves similar problems in an asynchronous setting. Our approach finds if there is an ordering between rendezvous of different channels based on the product machine. We believe that our algorithm works on a richer set of programs.

Lin [78] talks about an efficient compilation process of programs that have communication constructs similar to SHIM. He uses Petri nets to model the program and uses loop unrolling techniques. We did not attempt this approach because loop unrolling would cause the state space to explode even for small SHIM programs.

Static verification methods already exist for SHIM. In Chapter 7, we built a synchronous system to find deadlocks in a SHIM program. We make use of the fact that for a particular input sequence, if a SHIM program deadlocks under one schedule it will deadlock under any other. By contrast, the property we check in this chapter is not schedule independent: two channels may rendezvous at the same time under one schedule but may not under another schedule. This makes our problem more challenging.

There is a partial evaluation method [48] for SHIM that combines multiple concurrent processes to produce sequential code. Again, the work makes use of the scheduling independence property by expanding one task at a time until it terminates or blocks on a channel. On the other hand, in this chapter, we expand all possible communications from a given state forcing us to consider all tasks that can communicate from that state, rather than a single task.

11.7 Conclusions

We presented a static buffer memory minimization technique for the SHIM concurrent language. We obtain the partial order between communication events on channels by forming the product machine representing the behavior of all tasks in a program. To avoid state space explosion, we can treat the program as consisting of separate pieces.

We remove bounded recursion and expand each SHIM program into a tree of tasks and use sound abstractions to construct for each task an automaton that performs communication. Then we use the merging rules to combine tasks.

We abstract away data and computation from the program and only maintain parallel, communication and branch structures. We abstract away the data-dependent decisions formed by conditionals and loops and do not differentiate between scheduling choices and conditional branches. This may lead to false positives: our technique can discard pairs even though they can share buffers. However, our experimental results suggest this is not a big disadvantage and in any case our technique remains safe.

Our algorithm can be practically applied to the SHIM compiler that generates code for the Cell Broadband Engine. We found we could save 344KB of the PPE's memory for an FFT example.

We reduce memory without affecting the runtime schedule or performance. By sharing, two or more buffer pointers point to the same memory location. This can be done at compile time during the code-generation phase.

Chapter 12

Optimizing Barrier Synchronization

The SHIM model uses channels for communication. Channels generally carry data, but dataless channels are also interesting because they act as barriers. Dataless channels are used merely for synchronization and are popularly known as clocks in many programming languages.

In this chapter, we improve the runtime efficiency of clocks that are restricted versions of SHIM's channels. Like SHIM's channels, clocks are usually implemented using primitive communication mechanisms and thus spare the programmer from reasoning about low-level implementation details such as remote procedure calls and error conditions.

We statically analyze the use of these clocks—a form of synchronization barriers—in the Java-derived X10 concurrent programming language [29; 106] and use the results to safely substitute more specialized implementations of these standard library elements. X10's clocks were motivated from SHIM's channels and we believe that this analysis can also be applied in the SHIM setting.

A clock in X10 is a structured form of synchronization barrier useful for expressing patterns such as wavefront computations and software pipelines. Concurrent tasks registered on the same clock advance in lockstep. This is analogous to concurrent tasks in SHIM registered with the same channel.

Clocks provide flexibility, but programs often use them in specific ways that do not require their full implementation. In this chapter, we describe a tool that mitigates the overhead of general-purpose clocks by statically analyzing how programs use them and choosing optimized implementations when available.

Our static analysis technique models an X10 program as a finite automaton; we ignore data but consider the possibility of clocks being aliased. We pass this automaton to the NuSMV model checker [34], which reports erroneous usage of a clock and whether a particular clock follows certain idioms. If the clocks are used properly, we use the idiom information to restructure the program to use a more efficient implementation of each clock. The result is a faster program that behaves like one that uses the general-purpose library.

Our analysis flow has been designed to be flexible and amenable to supporting a growing variety of patterns. In the sequel, we focus on inexpensive queries that can be answered by treating programs as sequential. While analysis time is negligible, speedup is considerable and varies across benchmarks from a few percent to a 3× improvement in total execution time.

The techniques we present can be applied to a large class of concurrent languages, not just X10 or SHIM. These kind of optimizations are very useful when a bunch of programs follow a certain pattern and can be specialized.

In summary, our contributions are

- a methodology for the analysis and specialization of clocked programs;
- a set of cost-effective clock transformations;
- a prototype implementation: a plug-in for the X10 v1.5 tool chain; and
- experimental results on some modest-size benchmarks.

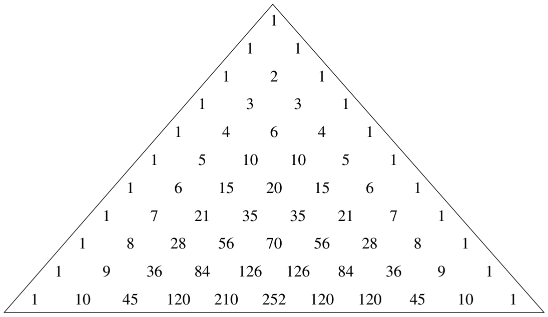
After a brief overview of the X10 language in Section 12.1 and the clock library in Section 12.2, we describe our static analysis technique in Section 12.3 and how we use its results to optimize programs in Section 12.4. We present experimental evidence that our technique can improve the performance of X10 programs in Section 12.5. We discuss related work in Section 12.6 and conclude in Section 12.7.

12.1 The X10 Programming Language

X10 [29; 106] is a parallel, distributed object-oriented language. To a Java-like sequential core it adds constructs for concurrency and distribution through the concepts of *activities* and *places*. An activity is a unit of work, like a thread in Java; a place is a logical entity that contains both activities and data objects.

The X10 language is more flexible than SHIM. It allows races and does not impose hard restrictions on how activities should be created. The *async* construct creates activities; parent and child execute concurrently. The X10 program in Figure 12.1 uses clocks to recursively compute the first ten rows of Pascal's Triangle. The call of the method *row* on line 40 creates a new stream object, spawns an

```
public class IntStream {
  public final clock clk = clock.factory.clock(); // stream clock
  private final int[] buf = new int[2]; // current and next stream values

  public IntStream(final int v) {
    buf[0] = v; // set initial stream value
  }

  public void put(final int v) {
    clk.next(); // enter new clock phase
    buf[(clk.phase()+1)%2] = v; // set next stream value
    clk.resume(); // complete clock phase
  }

  public int get() {
    clk.next(); // enter new clock phase
    final int v = buf[clk.phase()%2]; // get current stream value
    clk.resume(); // complete clock phase
    return v;
  } }

public class PascalsTriangle {
  static IntStream row(final int n) {
    final IntStream r = new IntStream(1); // start row with 1
    async clocked(r.clk) { // spawn clocked task to compute row's values
      if (n > 0) { // recursively compute previous row
        final IntStream previous = row(n-1);
        int v; int w = previous.get();
        while (w != 0) {
          v = w; w = previous.get();
          r.put(v+w); // emit row's values
        }
      }
      r.put(0); // end row with 0
    }
    return r;
  }

  public static void main(String[] args){
    final IntStream r = row(10);
    int w = r.get(); // print row excluding final 0
    while (w != 0) { System.out.println(w); w = r.get(); }
  } }
```



```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        1   6  15  20  15   6   1
      1   7  21  35  35  21   7   1
    1   8  28  56  70  56  28   8   1
  1   9  36  84 126 126  84  36   9   1
1  10  45 120 210 252 120 120  45  10   1
```

Figure 12.1: A program to compute Pascal's Triangle in X10 using clocks

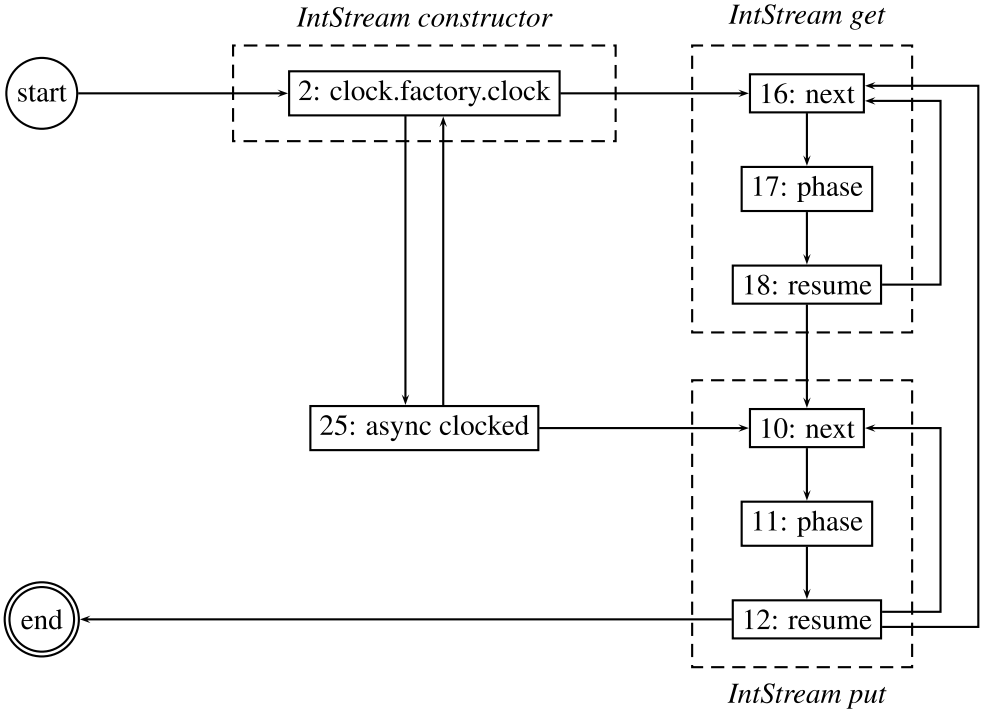


Figure 12.2: The automaton model for the clock in the Pascal's Triangle example

activity to produce the stream values, and finally returns the stream object to *main*. The rest of *main* executes in parallel with the spawned activity, printing the stream values as they are produced.

Spawned activities may only access final variables of enclosing activities, e.g.,

```
final int a = 3; int b = 4;
async { int x = a;        // OK: a is a final
        int y = b; }      // ERROR: b is not final
```

An X10 program runs in a fixed, platform-dependent set of places. The *main* method always runs in *place.FIRST_PLACE*; the programmer may specify where other activities run. Activities cannot migrate between places.

```
final IntStream s = new IntStream(4);
async (place.LAST_PLACE) { // spawn activity at place.LAST_PLACE
       // cannot call methods of s if LAST_PLACE != FIRST_PLACE
       final int i = 3;
       async (s) s.put(i); // spawn activity at the place of s; s is local => ok to deref
}
```

Activities that share a place share a common heap. While activities may hold references to remote objects, they can only access the fields and methods of a remote object by spawning an activity at the object's place.

X10 also introduces *value classes*, whose fields are all *final*. The fields and methods of an instance of a value class may be accessed remotely, unlike normal classes. Clocks are implemented as value classes.

X10 provides two primitive constructs for synchronization: *finish* and *when*. *finish p q* delays the execution of statement *q* until after statement *p* and all activities recursively spawned by *p* have completed. For example,

```
finish { async { async { System.out.print(``Hello''); } } }
System.out.println(`` world'');
```

prints "Hello world." The statement *when*(e) *p* suspends until the Boolean condition *e* becomes true, then executes *p* atomically, i.e., as if in one step during which all other activities in the same place are suspended.[1]

X10 also permits unconditional atomic blocks and methods, which are specified with the *atomic* keyword. For example,

```
atomic { int tmp = x; x = y; y = tmp; }
```

12.2 Clocks in X10

Clocks in X10 are a generalization of barriers. Unlike X10's *finish* construct, clocks permit activities to synchronize repeatedly. By contrast to *when* constructs, they provide a structured, distributed, and determinate form of coordination. While a complete discussion of X10's clocks is beyond the scope of this chapter, the

[1] X10 does not guarantee that *p* will execute if *e* holds only intermittently.

following subsections will demonstrate that clocks are amenable to efficient and effective static analysis.

Figure 12.3 lists the main elements of the clock API. An activity must be registered with a clock to interact with it. Activities are registered in one of two ways: creating a clock with the *clock.factory.clock()* static method automatically registers the calling activity with the new clock. Also, an activity can register activities it spawns with the *async clocked* construct.

```
final clock clk = clock.factory.clock();
async clocked(clk) { A1; clk.next(); A2; clk.next(); A3 }
async clocked(clk) { B1; clk.next(); B2; }
async { C; }
M1; clk.resume(); M1_2; clk.next(); M2;
```

A clock synchronizes the execution of activities through phases. A registered activity can request the clock to enter a new phase with a call to *next*, which blocks the activity until all other registered activities are done with the current phase, i.e., have called *next* or *resume*. For instance, in the program above, action A1 must complete before action B2 can start. In other words, A1 and B1 belong to phase 1 of clock *clk*; A2 and B2 belong to phase 2. C, however, does not belong to an activity registered with *clk*; it may execute at any time.

The *resume* method provides slack to the scheduler.[2] An activity calls *resume* when it is done with the current clock phase but does not yet need to enter the next. Unlike *next*, *resume* does not block the activity, and the activity must still call *next* to enter the next phase. In the example above, while M1 must terminate before A2 can start and A1 must terminate before M2 can start, M1_2 may start before A1 completes and continue after A2 starts because of *resume*.

In Figure 12.1, the value at the pth column and nth row of this triangle ($0 \leq p \leq n$) is the number of possible unordered choices of p items among n. One task per row produces the stream of values for the row by summing the two entries from the row immediately above. Each stream uses a clock to enforce single-write-single-read interleaving, so each task registers with two clocks: its own and the clock for the row immediately above. The clocks ensure proper inter-row coordination.

The *phase* method returns the current phase index (counting from 1). Figure 12.1 demonstrates this and also how activities can register with multiple clocks (using recursion in this example).

Finally, activities can explicitly unregister from a clock by calling *drop*. Activities are implicitly unregistered from their clocks when they terminate.

The operations of an activity on a clock modify the state of this activity w.r.t. that clock. Figure 12.4 shows the behavior. The activity may be in one of four states: *Active*, *Resumed*, *Inactive*, or *Exception*. Transitions are labeled with clock-

[2] The *resume* method is typically used in activities registered with multiple clocks.

related operations: *async clocked*, *resume*, *next*, *phase*, and *drop*. For example, an activity moves from the *Active* state to *Resumed* if it calls *resume* on the clock. If it calls *resume* again, it moves to the *Exception* state. Any operation that leads to the *Exception* state throws the *ClockUseException* exception.

```
/* Create a new clock. Register the calling activity with this clock. */
final clock clk = clock.factory.clock();

/* Spawn an activity registered with clocks clk_1, ..., clk_n with body p. */
async clocked(clk_1, ..., clk_n) p

public interface clock {
        /* Notify this clock that the calling activity is done with whatever it intended
         * to do during this phase of the clock. Does not block. */
        void resume();

        /* Block until all activities registered with this clock are ready to enter the next
         * clock phase. Imply that calling activity is done with this phase of the clock. */
        void next();

        /* Return the phase index. Calling activity cannot be resumed on the clock. */
        int phase();

        /* Unregister the caller from this clock; release it from having to participate */
        void drop();
}
```

Figure 12.3: The clock API

12.2.1 Clock Patterns

We now describe the four clock patterns we currently identify. We believe that our techniques can also be applied to find other patterns.

Our first pattern is concerned with exceptions: can an activity reach the exception state for a particular clock? The default clock implementation looks for transitions to this state and throws *ClockUseException* if they occur. Aside from the annoyance of runtime errors, runtime checks slow down the implementation. We want to avoid them if possible.

Our algorithm finds that the clocks are used properly in the program of Figure 12.1; e.g., no task erroneously attempts to use a clock it is not registered with. Therefore, it substitutes the default implementation with one that avoids the overhead of runtime checks for these error conditions.

We also want to know whether *resume* is ever called on a clock. This feature's implementation requires additional data structures and slows down all clock

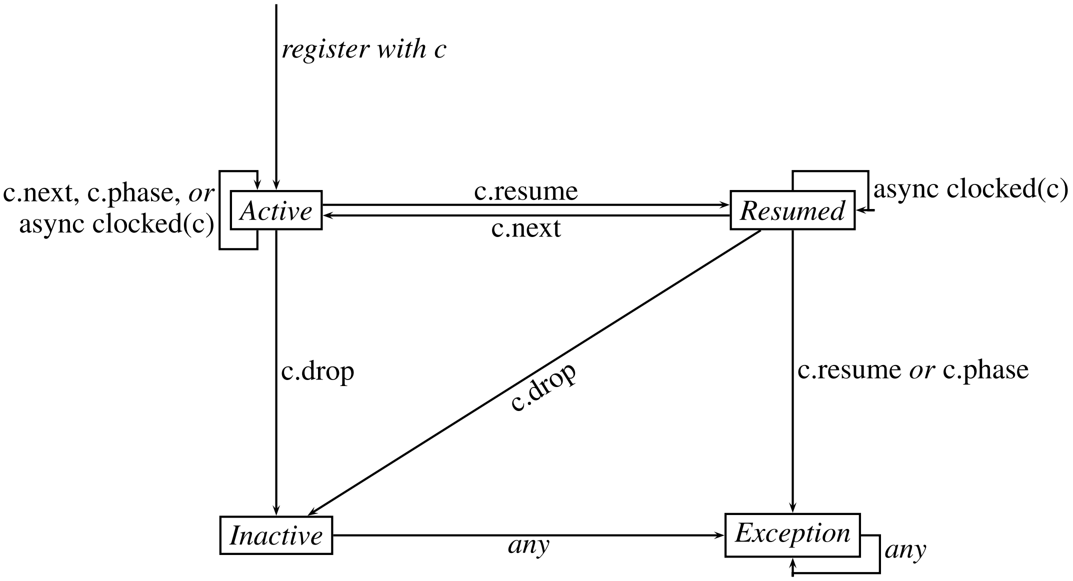


Figure 12.4: The state of one activity with respect to clock *c*

operations. We discuss this and other optimizations in Section 12.4.

Activities often use clocks to wait for subactivities to terminate. Consider

```
final clock clk = clock.factory.clock();
async clocked (clk) A1;
async A2;
async clocked (clk) A3;
clk.next();
A4;
```

Here, if A1 and A2 do not interact with clock *clk*, *clk.next()* requires activities A1 and A3 to terminate before A4 starts executing and nothing else. In particular, A2 and A4 may execute in parallel. We want to detect subactivities that are registered with the clock yet never request to enter a new clock phase.

Finally, the default clock implementation enables distributed activities to synchronize. If it turn out that all registered activities belong to the same place, a much faster clock implementation is possible. Our Pascal's Triangle program is a trivial example of this since all activities are spawned in the default place.

12.3 The Static Analyzer

In this subsection, we describe how we detect clock idioms. We start from the program's abstract syntax tree, compute its call graph, and run aliasing analysis on clocks. We then abstract data by replacing conditional statements with nondeterministic choice. From the control flow graph of this abstract program, we extract one automaton per clock. This gives a conservative approximation of the sequences of operations that the program may apply to the clock.

To a model checker, we feed the automaton for the control-flow of the program along with an automaton model of the clock API and a series of temporal logic properties, one for each idiom of interest. For each property and each clock, the model checker either proves the property or returns a counterexample in the form of a path in the automaton that violates the property.

We use the T.J. Watson Libraries for Analysis (WALA) [66] for parsing, call- and control-flow-graph construction, and aliasing analysis. We have extended the Java frontend of WALA to accommodate X10 and extract from the AST the required automata in the form of input files for the NuSMV model checker [34].

We now describe the key technical steps in detail. We start with the construction of the automaton, then discuss the encoding of the clock API, the temporal properties, and finally aliasing.

```
stmt1;
async clocked(c) {
  stmt3;
}
stmt2;
```

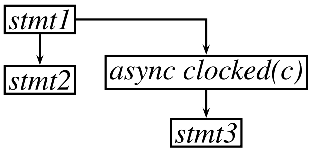


Figure 12.5: Modeling *async* calls

12.3.1 Constructing the Automaton

Figure 12.2 shows the automaton we build for the clock *clk* in Figure 12.1. Each operation on *clk* in the text of the program becomes one state, which we label with the type of operation and its line number. Transitions arise from our abstraction of the program's control flow. We highlighted the fragments corresponding to the constructor and methods of the *IntStream* class.

methods Each method body becomes a fragment of the automaton. Each call of a method adds a transition to and from its entry and exit nodes. For example, since *get* may be called twice in a row (lines 28 and 30), we added the edge from its exit node "18: resume" to its entry node "16: next." It may also be called after put, looping from line 31 back to line 30, so we added an edge from node "12: resume" to node "16: next."

conditionals We ignore guards on conditionals and add arcs for both branches. For example, the *if* on line 26 runs immediately after the *async clocked* on line 25. The "then" branch of this *if* runs line 27, which starts with a call to *row* that starts by constructing an *IntStream* (line 24) whose constructor calls *clock.factory.clock()* (line 2). This gives the arc from node "25: async clocked" to "2: clock.factory.clock." The "else" branch is line 34, which calls *put*, which starts with a call to *next* (line 10). This gives the arc to "10: next."

async Because we are not checking properties that depend on interactions among tasks, we can treat a spawned activity as just another path in the program. When execution reaches an *async* construct, we model it as either jumping directly to the task being spawned or skipping the child and continuing to execute the parent. This is illustrated in Figure 12.5.

In our Pascal's Triangle example, this means control may flow from the *IntStream* constructor exit point "2: clock.factory.clock" to the *async* construct "25: async clocked" or ignore the *async* and flow back via the *return* statement to the subsequent get method call in either *main* or *row*, i.e., node "16: next."

We build one automaton for each call of *clock.factory.clock* in the source code, meaning our algorithm does not distinguish clocks instantiated from the same allocation site. So we construct only one automaton for our example, even though the program uses ten (very similar) clocks when it executes.

We have taken a concurrent program and transformed it into a sequential program with multiple paths. Thanks to this abstraction, we avoid state space explosion both in the automaton construction and in the model checker.

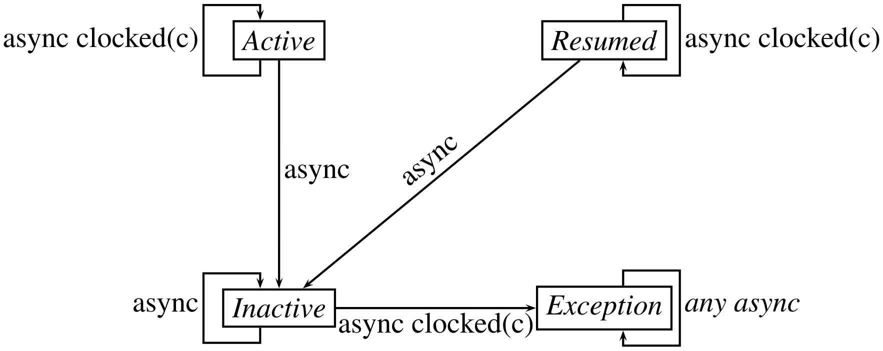


Figure 12.6: Additional transitions in the clock state for modeling *async* operations

12.3.2 Handling Async Constructs with the Clock Model

Our model of clock state transitions—Figure 12.4—only considers a single activity, but X10 programs may have many. As explained in Section 12.3.1, we model *async* constructs with nondeterministic branches, so we have to extend the typestate automaton (described later) for the clock to do the same.

Figure 12.6 shows the additional transitions necessary for handling *async* actions. We consider two cases: when analyzing clock *c* and we encounter *async clocked(c)*, the new activity stays either *Active* or *Resumed*. By contrast, if we encounter an *async* not clocked on *c*, the new activity starts in the *Inactive* state (arcs labeled just *async*).

12.3.3 Specifying Clock Idioms

Once we have the automata modeling the program and clock state, it becomes easy to specify patterns for NuSMV as temporal logic formulas.

Three patterns are CTL reachability properties of the form
SPEC AG`(!(`*target*`))`
where *target* is either the *Exception* state, a *resume* operation, or an *async clocked(c)* node annotated with a place expression, that is, a remote activity creation.

We check for the fourth pattern—whether spawned activities ever call *next* on the clock—by looking for control-flow paths that contain an *async clocked(c)* operation followed by a *c.next* operation. The LTL specification is
LTLSPEC G`(`*c_next* `->` ***H***`(!`*async_clocked_c*`))`

12.3.4 Combining Clock Analysis with Aliasing Analysis

Clocks can be aliased just like any objects. Figure 12.7 shows an example of aliasing of clocks in X10. We create two clocks *c1* and *c2*. The variable *x* can take the value of either *c1* or *c2* depending on the value of *n*.

We could abstract the program into two control paths, one that assumes $x = c1$ and one that assumes $x = c2$. However, this would produce a number of paths exponential in the number of aliases that have to be considered simultaneously.

Instead, we chose to bound the size of our program abstraction (at the expense of precision) as shown in the bottom three diagrams of Figure 12.7. We consider each clock operation on *x* in isolation and apply it nondeterministically to any of the possible targets of *x* as returned by WALA's aliasing engine.

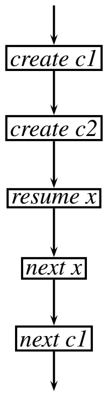
Figure 12.8 shows how we extend this idea to *async* constructs. Our tool reports that operations on clock *c1* cannot throw *ClockUseException*. However, it fails to establish the same for *c2* because our abstraction creates a false path—*next c2* following *async clocked(c1,c1)*.

12.4 The Code Optimizer

Results from our static analyzer drive a code optimizer that substitutes each instance of the clock class for a specialized version. We manually wrote an optimized version of the clock class for each clock pattern we encountered in our test cases; a complete tool would include more. Our specialized versions include a clock class that does not check for protocol violations (transitions to the *exception* state) and one that does not support *resume*.

There is one abstract clock base class that contains empty methods for all clock functions; each specialized implementation has different versions of these methods that uses X10 primitives to perform the actual synchronization. Our optimizer changes the code (actually the AST) to use the appropriate derived class for each clock, e.g., *c = clock.factory.clock()* would be replaced with *c = clock.factory.clockef()* if clock *c* is known to be exception free.

```
final clock c1 = clock.factory.clock();
final clock c2 = clock.factory.clock();
..
final clock x = (n > 1)? c1: c2;
x.resume();
x.next();
c1.next();
```



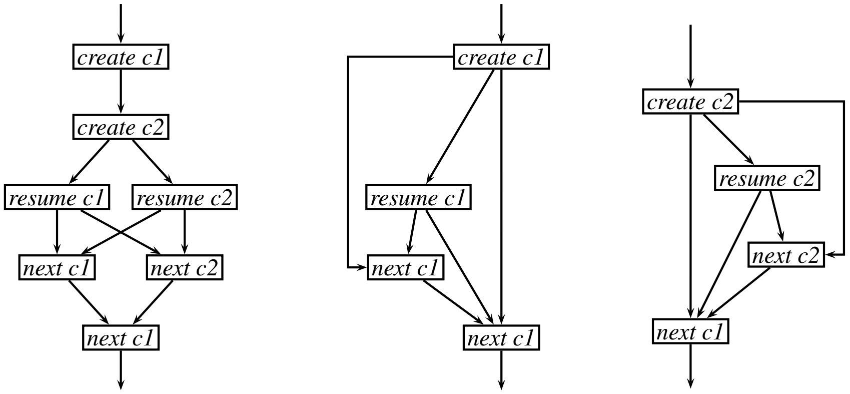
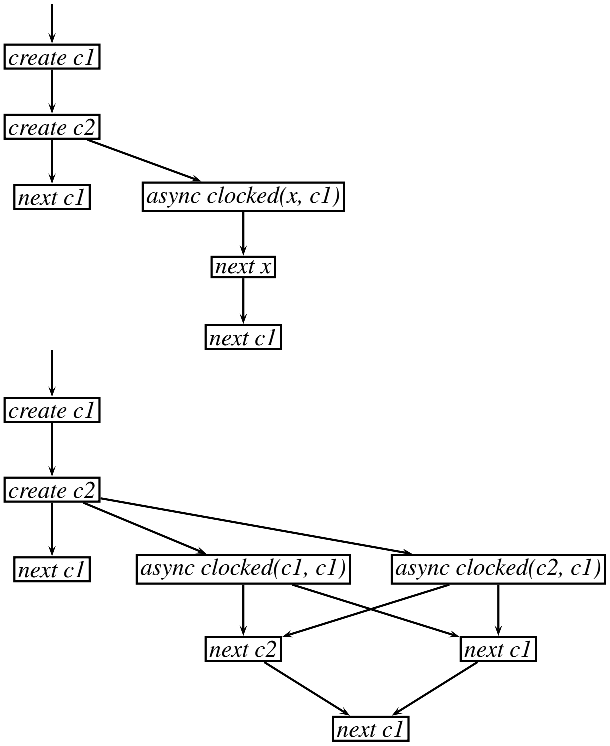


Figure 12.7: *Top Left*: Aliasing clocks in X10, *Top Right*: the corresponding control flow graph, *Bottom Left*: our abstraction, *Bottom Center*: automaton for *c1*, *Bottom Right*: automaton for *c2*

The top of Figure 12.9 shows the general-purpose implementation of *next*. The *clock* value class contains the public clock methods; the internal *ClockState* maintains the state and synchronization variables of the clock. The *next* method first verifies that the activity is registered with the clock (and throws an exception otherwise), then calls the *select* function to wait on a *latch*: a data structure that indicates the phase. The *latch* is either *null* if *next()* was called from an *active()* state or holds a value if *next()* was called from a *resumed()* state. The *wait* function blocks and actually waits for the clock to change phase. The *check* method decrements the number of activities not yet resumed on the clock and advances the clock phase when all activities registered on the clock are resumed.

A basic optimization: when we know the clock is used properly, we can elim-



```
final clock c1 =
  clock.factory.clock();
final clock c2 =
  clock.factory.clock();
..
final clock x = (n > 1)? c1: c2;
async clocked(x, c1) {
  x.next();
  c1.next();
}
c1.next();
```

Figure 12.8: Asyncs and Aliases

inate the registration check in *next* and elsewhere. Figure 12.9 shows such an exception-free implementation.

Accommodating *resume* carries significant overhead, but if we know the *resume* functionality is never used, we can simplify the body of *select* as shown in Figure 12.9. We removed the now-unneeded *latch* object and can do something similar in other methods (not shown).

Figure 12.9 also shows a third optimization. Because clocked activities may be distributed among places, synchronization variables have to be updated by remote activities. When we know a clock is only used in a single place, we dispense with the *async* and *finish* constructs.

```
// The default implementation

class ClockState {
..
  atomic int check() {
    int resumedPhase = currentPhase;
    if (remainingActivities-- == 0) {
       // set the number of activities
       // expected to resume
       remainingActivities =
           registeredActivities;
       // advance to the next phase
       currentPhase++;
    }
    return resumedPhase;
  }

 void wait(final int resumedPhase) {
     when(resumedPhase != currentPhase);
  } }
```

```
value class clock {
..
  final ClockState state = new ClockState();
..
  void select(nullable<future<int>> latch) {
    if (latch == null) {
      async (state) state.wait(state.check());
    } else {
      final int phase = latch.force();
      async (state) state.wait(phase);
    }
  }

  public void next() {
    if (!registered())
        throw new ClockUseException();
    finish select(ClockPhases.put(this, null));
  } }
```

```
// An exception-free implementation

public void next() {
  finish
    select(ClockPhases.put(this, null));
}
```

```
// For when resume() is never used

void select() {
  async (state) state.wait(state.check());
}

public void next() {
  if (!registered())
      throw new ClockUseException();
  finish select();
}
```

```
// A clock is only in a single place

void select(nullable<future<int>> latch) {
  if (latch == null)
    state.wait(state.check());
  else
    state.wait(latch.force());
}

public void next() {
  if (!registered())
      throw new ClockUseException();
  select(ClockPhases.put(this, null));
}
```

Figure 12.9: Various implementations of *next* and related methods

Table 12.1: Experimental Results of our clock specialization

Example	**Clocks**	**Lines**	**Result**	**Speed**	**Analysis Time**	
				Up	**Base**	**NuSMV**
Linear Search	1	35	EF, NR, L	35.2%	33.5s	0.4s
Relaxation	1	55	EF, NR, L	87.6	6.7	0.3
All Reduction Barrier	1	65	EF, NR	1.5	27.2	0.1
Pascal's Triangle	1	60	EF, L	20.5	25.8	0.4
Prime Number Sieve	1	95	NR, L	213.9	34.7	0.4
N-Queens	1	155	EF, NR, ON, L	1.3	24.3	0.5
LU Factorization	1	210	EF, NR	5.7	20.6	0.9
MolDyn JGF Bench.	1	930	NR	2.3	35.1	0.5
Pipeline	2	55	Clock 1: EF, NR, L Clock 2: EF, NR, L	31.4	7.5	0.5
Edmiston	2	205	Clock 1: NR, L Clock 2: NR, L	14.2	29.9	0.5

EF: No ClockUseException, NR: No Resume, ON: Only the activity that created the clock calls *next* on it, L: Clocked used locally (in a single place)

12.5 Results

We applied our static analyzer to various programs, running it on a 3 GHz Pentium 4 machine with 1 GB RAM. Since we want to measure the overhead of the clock library, we purposely run our benchmarks on a single-core processor. Table 12.1 shows the results. For each example, we list its name, the number of clock definitions in the source code, its size (number of lines of code, including comments), what our analysis discovered about the clock(s), how much faster the executable for each example ran after we applied our optimizations, and finally the time required to analyze the example. (The *Base* column includes the time to read the source, build the IR, perform pointer analysis, build the automata, etc.; *NuSMV* indicates the time spent running the NuSMV model checker. Total time is their sum.)

The first example is a paced linear search algorithm. It consists of two tasks that search an array in parallel and use a clock to synchronize after every comparison. The Relaxation example, for each cell in an array, spawns one activity that repeatedly updates the cell value using the neighboring values. It uses a clock to force these activities to advance in lockstep. The All Reduction Barrier example is a variant on Relaxation that distributes the array across multiple places. Pascal's

Triangle is the example of Figure 12.1. Our prime number sieve uses the Sieve of Eratosthenes. N-Queens is a brute-force tree search algorithm that uses a clock to mimic a join operation. LU Factorization decomposes a matrix in parallel using clocks. We also ported the MolDyn Java Grande Forum Benchmark [111] in X10 with clocks, the largest application on which we ran our tool. Pipeline has three stages; its buffers use two clocks for synchronization. Edmiston aligns substrings in parallel and uses two clocks for synchronization.

The Result column lists the properties satisfied by each example's clocks. For example, the N-Queens example cannot throw *ClockUseException*, does not call *resume*, and uses only locally created clocks. Our tool reports the JGF benchmark may throw exceptions and pass clocks around, although it also does not call *resume*. In truth, it does not throw exceptions, but our tool failed to establish this because of the approximations it uses. This reduced the speedup we could achieve, but does not affect correctness.

The Linear Search, Relaxation, Prime Number Sieve, and Pipeline examples use clocks frequently and locally, providing a substantial speedup opportunity. Although our analysis found N-Queens satisfies the same properties as these, we could improve it up only slightly because its clock is used rarely and only in one part of the computation. Switching to the local clock implementation provided the majority of the speedup we observed, but our 5% improvement on the already heavily optimized distributed LU Factorization example is significant.

Our tool analyzed each example in under a minute and the model checker took less than a second in each case. Most of the construction time is spent in call- and control-flow graph constructions and aliasing analysis, which are already done for other reasons, so the added cost of our tool is on the order of seconds, making it reasonable to include as part of normal compilation.

12.6 Related Work

Typestate analysis [114] tracks the states that an object goes through during the execution of a program. Standard typestate analysis and concurrency analysis are disjoint. Our analysis can be viewed as a typestate analysis for concurrent programs. Clocks are shared, stateful objects. We therefore have to track the state of each clock from the point of view of each activity.

Model checking concurrent programs [36; 34] is usually demanding because of the potential for exponentially large state spaces often due to having to consider different interleavings of concurrent operations. By contrast, our technique analyzes concurrent programs as if they were sequential—we consider spawned tasks to be additional execution paths in a sequential program—hence avoiding the explosion.

Concurrency models come in many varieties. We showed in Chapter 7 that the state space explosion can also be avoided by carefully choosing the primitives of the concurrent programming language. Unfortunately, this restricts the flexibility of the language. Our work focuses on concurrency constructs similar to those advocated by us in Chapter 7, but features like resume and aliased clocks are absent from their proposal. We trade a more flexible concurrency model against the need for further approximation in modeling the programs.

Static analysis of concurrency depends greatly on the underlying model. Although X10 supports both message-passing-style and shared-memory-style concurrency (in the case of co-located activities), we focus exclusively on its message-passing aspects, as have others. Mercouroff [84] approximates the number of messages between tasks in CSP [62] programs. Reppy and Xiao [102] analyze communication patterns in CML. Like ours, their work aims at identifying patterns amenable to more efficient implementations. They attempt to approximate the number of pending send and receive operations on a channel. Our work is both more specific—it focuses on clocks—and more general: our tool can cope with any CTL or LTL formula about clock operations.

Reppy and Xiao use modular techniques; we consider an X10 program as a whole. A modular approach may improve out tool's scaling, but we have not explored this yet.

Analysis of X10 programs has also been considered. Agarwal et al. [2] describe a novel algorithm for may-happen-in-parallel analysis in X10 that focuses on atomic subsections. Chandra et al. [27] introduce a dependent type system for the specification and inference of object locations. We could use the latter to decide whether activities and clocks belong to the same place.

12.7 Conclusions and Future Work

We presented a static analysis technique for clocks in the X10 programming language. The result allows us to specialize the implementation of each clock, which we found resulted in substantial speed improvements on certain benchmark programs. Our technique has the advantage of being able to analyze a concurrent language using techniques for sequential code.

We treat each clock separately and model subtasks as extra paths in the program, much like conditionals. We abstract away conditional predicates, which simplifies the structure at the cost of introducing false positives. However, our technique is safe: we revert to the unoptimized, general purpose clock implementation when we are unsure a particular property is satisfied. Adding counter-example guided abstraction refinement [37] could help.

We produce two automata for each clock: one models the X10 program; the other encodes the protocol (typestate) for the clock. We express the automata in a form suitable for the NuSMV model checker. Experimentally, we find NuSMV is able to check properties for modestly sized examples in seconds, which we believe makes it fast enough to be part of the usual compilation process.

Finally, we plan to extend these ideas to SHIM — provide verification based specialization of the generated code and see how the efficiency improves.

Chapter 13

Optimizing Locks

In the previous chapters, we optimized deterministic constructs at the language level. Most of these constructs are implemented using low-level constructs such as locks. In this chapter, we improve the efficiency of locks, especially when there is biased behavior. We do not directly solve the nondeterminism problem here, but a general concurrency problem that can be applied to any system that uses locks.

Locks are used to ensure exclusive access to shared memory locations. Unfortunately, lock operations are expensive, so work has been done on optimizing their performance for common access patterns. One such pattern is found in networking applications, where there is a single thread dominating lock accesses. An important special case arises when a single-threaded program calls a thread-safe library that uses locks.

Another instance occurs when a channel is used dominantly by a single thread in the D^2C model (Chapter 10) that allows multiple writes but in a deterministic way. Shared variables and communication are implemented using locks. In such cases, we want the dominant thread to access the channel in an efficient way and thus we would like to optimize locks that constitute a major component in the implementation of a channel.

An effective way to optimize the dominant-thread pattern is to “bias” the lock implementation so that accesses by the dominant thread have negligible overhead. We take this approach in this work: we simplify and generalize existing techniques for biased locks, producing a large design space with many trade-offs. For example, if we assume the dominant process acquires the lock infinitely often (a reasonable assumption for packet processing), it is possible to make the dominant process perform a lock operation without expensive fence or compare-and-swap instructions. This gives a very low overhead solution; we confirm its efficacy by experiments. We show how these constructions can be extended for lock reservation, re-reservation, and to reader-writer situations.

13.1 Introduction

Programmers typically use locks to control access to shared memory and to achieve determinism. While using locks correctly is often the biggest challenge, programmers are also concerned with their efficiency. We are too: this work improves the performance of locking mechanisms by using knowledge of their access patterns to speed the common case.

Figure 13.1 shows the standard way of implementing a spin-lock using an atomic compare-and-swap (CAS) operation. To acquire the lock, a thread first waits (“spins”) until the lock variable *lck* is 0 (indicating no other thread holds the lock), then attempts to change the lock value from 0 to 1. Since other threads may also be attempting to acquire the lock at the same time, the change is done atomically to guarantee only one thread changes the value. Although other threads’ *while* loops would see the lock variable become 0, their compare-and-swap would fail because the winning thread would have changed the lock to 1.

```
void lock(int *lck) {
 bool success;
 do {
    while (*lck != 0) {} /* wait */
    success = compare_and_swap(lck, 0, 1);
  } while (!success);
}

void unlock(int *lck) { *lck = 0; }

atomic /* function is one atomic machine instruction */
bool compare_and_swap(int *lck, int old, int new) {
  if (*lck == old) {
    *lck = new; return 1;
  } else
    return 0;
}
```

Figure 13.1: A spin lock using atomic compare-and-swap

We found, on an unloaded 1.66 GHz Intel Core Duo, the compare-and-swap instruction took seven times longer than “*counter*++,” a comparable nonatomic read-modify-write operation. The cost when there is contention among multiple processors can be substantially higher, especially if a cache miss is involved. This overhead can be prohibitive for a performance-critical application such as packet processing, which may have to sustain line rates of over 1 Gbps and thus has a very limited cycle budget for actual processing. Reducing locking overhead, therefore, can be very useful.

Bacon et al.'s thin locks for Java [8] are an influential example of lock optimization. Their technique was motivated by the observation that sequential Java programs often needlessly use locks indirectly by calling thread-safe libraries. To reduce this overhead, thin locks overlay a compare-and-swap-based lock on top of Java's more costly monitor mechanism. Thus a single-threaded program avoids all monitor accesses yet would operate correctly (i.e., use monitors) if additional threads were introduced. Thin locks considerably reduce overhead but still require one atomic operation per lock acquisition.

A refinement of this technique [71; 94; 7] further improves performance by allowing a single thread to reserve a lock. Acquisitions of the lock by the reserving thread do not require an atomic operation but do require the part-word technique that achieves the same functionality as fences with almost the same cost.

Lamport [74] also optimizes for the low contention access pattern by avoiding atomic operations. This algorithm uses a bakery-style algorithm to resolve contention, which has been found to be less efficient than algorithms that do use atomic operations, such as the MCS lock [83].

Lopsided lock-access patterns in network packet-processing applications motivated our work. In a typical architecture, a packet is read off a network card by a dedicated core and then dispatched to one of several processing cores. In the commercial network-traffic analyzer with which we are familiar, the packets are partitioned among cores by source address; i.e., all packets with the same source address are sent to the same core. Each processing core maintains data structures for its group of source addresses. Nearly all access to a group is from the owner core. Occasionally, however, a core might update information for a group held by a different core; thus, it is necessary to maintain atomicity of updates using locks. Such an arrangement of data and processing results in a highly biased access pattern for a data item: the owner is responsible for a large (90% or more) fraction of the accesses to its data, the rest originate from other cores.

This work looks at the question of optimizing lock performance under such lopsided access patterns. It makes four contributions. First, we provide a generic method for building biased locks. In a nutshell, we implement biased locks with a two-process mutual exclusion algorithm between the dominant thread and a single representative of all of the other threads, chosen with a generic N-process mutual exclusion algorithm. This construction simplifies and generalizes the algorithm of Kawachiya et al. [94], which is a specific combination of this type that intertwines a Dekker-lock for two threads and a CAS-based lock for an N-thread mutex. Our experiments show that different choices for the N-process mutex algorithm can improve overall performance.

Our second contribution is a simple scheme for changing the primary owner of a lock (“re-reservation”). The scheme given by Kawachiya et al. [71] is heavy-weight: it requires suspending the thread owning the lock and often modifying its program counter to a retry point; in their later work, they abandoned it for this reason [94]. By contrast, we show a simple way to change a lock’s owner without suspending the existing owner.

In our third contribution, we establish conditions under which atomic and memory fence operations in a dominant thread can be *dispensed with entirely*. Most multiprocessor memory systems do not provide sequential consistency across threads: a sequence of writes by one thread may appear to occur in a different order to a different thread. Few synchronization algorithms can cope with such an unruly communication mechanism, so multiprocessors typically provide costly but effective “fence” instructions that force all outstanding writes to complete. Experiments on the Intel Core Duo chip show that their “mfence” instructions require about two to three clock cycles. We show memory fences are essential for the biased lock construction described above, assuming the weaker memory ordering imposed by store-buffer forwarding, which is a feature of most modern processors. We prove that for a processor with store-forwarding, any mutual exclusion algorithm with a “symmetric choice” property requires memory fences. The symmetric choice property is that there is a protocol state where either of two contending threads may acquire the lock. Since standard algorithms such as those by Dekker [45], Peterson [96], and Lamport [74] have the symmetric choice property, they all require memory fences to be correct. Our proposed solution, therefore, is asymmetric by nature: it requires the dominant thread to grant access to the lock after receiving a request from a nondominant thread. The protocol as a whole is free from starvation provided the dominant thread checks for such requests infinitely often.

Finally, we introduce biased read-write locks. A read-write lock allows multiple readers to read at the same time, but only one writer to access the critical section at any time. We show, along with experiments that the general construction of bias in normal locks can be extended to provide biased read-write locks.

In summary, we make four new contributions in this chapter:

1. we provide a simple, generalized construction of biased locks (Kawachiya [71] is a special case of our algorithm);
2. we provide a light-weight scheme for changing the owner of a lock dynamically;
3. we introduce asymmetric locks; and
4. we apply bias to read-write locks.

In the next section, we describe our generic owner-based locking scheme, which assumes a fixed owner. We then discuss the algorithm for switching ownership (Section 13.3). The formalization of memory fence operations, the symmetric choice property, and the subsequent proofs are discussed in Section 13.4. We define asymmetric locks in Section 13.5. We discuss how we verified our algorithms in Section 13.7 and discuss experimental results in Section 13.8.

13.2 Flexible, Fixed-Owner Biased Locks

In this section, we define a flexible biased locking scheme that assumes a lock is owned by a fixed, pre-specified thread. The scheme reduces the cost of access for the owning thread. In particular, the scheme does not incur the cost of a compare-and-swap operation, but it does require memory fences for correctness. From this point on, we focus on the x86 architecture; the kind of fences and their placement may differ for other architectures.

At its core, our scheme employs different locking protocols for the owner and the nonowners. For the owner, any two-process mutual exclusion protocol with operations *lock2* and *unlock2* suffices; for the other threads, we use a generic *N*-process mutual exclusion protocol with operations *lockN* and *unlockN*. This exploits complementary characteristics: protocols that rely only on atomicity of read and write, such as Peterson's algorithm [96], are efficient for two processes but not necessarily for larger numbers of threads; protocols based on atomic primitives, such as the MCS lock [83], are more effective when there are many contending threads.

Figure 13.2 shows our biased lock scheme. The *this_thread_id* identifier contains a unique number identifying the current thread. The nonowner threads first compete for the *N*-process lock; the winning thread then competes for the two-process lock with the owner process.

It is easy to see the scheme assures mutual exclusion among the threads provided the two locking protocols work, and thread IDs are well-behaved; other properties depend on the locking protocols themselves. For example, the combined protocol is starvation free if both locking protocols are; if only the 2-process locking protocol is starvation free, the owner is always guaranteed to obtain the lock but one or more of the nonowning threads could remain forever in the waiting state. Similar results hold for bounded waiting, assuming starvation freedom.

This scheme can be implemented by employing Dekker's algorithm for 2-process locking and the compare_and_swap spin-lock algorithm from the introduction for *N*-process locking. Such an implementation is similar to Onodera et al. [94], but differs in the details of how *N*-process locking is invoked.

An alternative: use Peterson's algorithm (Figure 13.3) for 2-process locking and the MCS algorithm for *N*-process locking. On the Intel architecture, Peterson's

```
typedef struct {
  ThreadId owner;
  Lock2 t; /* lightweight, 2-process lock */
  LockN n; /* N-process lock */
} Lock;

biased_lock(Lock *l) {
  if (this_thread_id == l->owner)
    lock2(l->t);
  else {
    lockN(l->n);
    lock2(l->t);
  }
}

biased_unlock(Lock *l) {
  if (this_thread_id == l->owner)
    unlock2(l->t);
  else {
    unlock2(l->t);
    unlockN(l->n);
  }
}
```

Figure 13.2: Our general biased-lock scheme

algorithm requires memory fences to ensure operations issued before the fence are carried out before operations issued after the fence and to ensure that updates to shared variables are made visible to other threads. This is because newer x86 implementations employ "store-forwarding" that effectively propagates memory updates lazily, depositing them in processor-local store buffer before ultimately dispatching them to the memory system. Hence the store buffer functions as an additional level of cache and improves performance.

Unfortunately, store buffers break sequential memory consistency between processors. To ensure local sequential consistency, a processor always consults its local store buffer on a read to ensure it sees all its earlier writes, but the contents of each processor's (local) store buffer are not made visible to other processors, meaning a shared memory update may be delayed or even missed by other processors. For instance, if variables x and y are both initialized to 0, one thread executes *write x 1; read y*, and another thread executes *write y 1; read x*, it is possible under store forwarding for both threads to read 0 for both x and y, an outcome that is impossible under sequential consistency. Intel's reference manual [67] provides more details and examples.

In the protocol in Figure 13.3, in the absence of the first fence, thread i may not see the updated flag value of thread j and thread j may not see the updated flag value of thread i. This would allow both threads to enter the critical section at once, violating mutual exclusion. The second fence ensures all changes to global variables made in the critical section become visible to other processors.

```
flag[i] = 1;
turn = j;
fence(); /* force other threads to see flag and turn */
while (flag[j] && turn == j) {} /* spin */
/* ...critical section... */
fence(); /* make visible changes made in critical section */
flag[i] = 0;
```

Figure 13.3: Peterson's mutual exclusion algorithm for process i when running concurrently with process j. Its correctness demands memory fences.

13.3 Transferring Ownership On-The-Fly

Our biased lock scheme from the last section assumes that the dominant thread is fixed and known in advance. However, certain applications may need to change a lock's dominant thread, such as when ownership of shared data is passed to a different thread. We call this ownership transfer or re-reservation. In this section, we describe a simple method for effecting this transfer. Figure 13.4 shows the outline of our method. We do not fix a particular condition for switching ownership—each application may define its own condition for when a switch is necessary. One such scheme, for instance, is to maintain an average frequency of usage of a lock by each thread, and switch ownership when the frequency of a nondominant thread exceeds that of the dominant one.

The bias-transfer mechanism necessarily switches the status of a nondominant thread. There are certain times when doing so is not safe. For example, it would be incorrect to do so when the dominant thread is about to enter its critical section, so we require that a nondominant thread hold the biased lock before switching its status to dominant. This requirement is not, however, sufficient in itself. A thread may switch to being dominant at a point in time where the earlier dominant thread (line 12) is waiting for its lock. Therefore, we demand additional synchronization between the old and new dominant threads.

The *try* flag array (line 5), which has one entry per thread, provides synchronization. If thread A is dominant, the *try*[A] entry, if set, indicates to other threads that the owner *may be* in the process of acquiring the lock in lines 9–12. Meanwhile, if some other thread (say, B) calls *switch_to_dominant* in an attempt to become dominant (lines 29–37), then B changes the owner and waits for the

```
 1 typedef struct {
     ThreadId owner;
 3   Lock2 t; /* lightweight, 2-process lock */
     LockN n; /* N-process lock */
 5   bool try[NTHREADS];
   } Lock;
 7
   void biased_lock(Lock *l) {
 9   l->try[this_thread_id] = 1;
     fence();
11   if (this_thread_id == l->owner) {
       lock2(l->t);
13     if (this_thread_id != l->owner) {
         /* owner has changed */
15       unlock2(l->t);
         goto NON_OWNER;
17     } else /* owner has not changed */
         l->try[this_thread_id] = 0;
19   } else {
     NON_OWNER:
21     l->try[this_thread_id] = 0;
       lockN(l->n);
23     lock2(l->t);
     }
25 }
```

```
28 void switch_to_dominant(Lock *l)
   {
30   lockN(l->n);
     lock2(l->t);
32   prev_owner = l->owner;
     l->owner = this_thread_id;
34   unlock2(l->t);
     while (l->try[prev_owner]) {}
36   unlockN(l->n);
   }
38
   void biased_unlock(Lock *l){
40   if (this_thread_id == l->owner)
       unlock2(l->t);
42   else {
       unlock2(l->t);
44     unlockN(l->n);
     }
46 }
```

Figure 13.4: Bias Transfer

previously dominant thread *A* to reach a stable state: one where it is certain that *A* realizes the change of ownership (line 35).

This procedure adds a few instructions (starting line 9) to the lock algorithm for the owner thread. The overhead is two additional assignments, one test and a fence instruction, due to the infrequency of owner switching and the expected infrequency in nonowner locks.

13.4 Mutual Exclusion and Memory Fences

Given the high cost of atomic and fence operations, one may wonder whether there are mutual exclusion schemes where these operations are not needed. Classical algorithms such as Dekker or Peterson do not use atomic operations, but do require fences to be correct on modern architectures. In this section, we show that the use of fences is unavoidable if the architecture supports store-buffer forwarding unless certain requirements are relaxed.

Fence and atomic operations have the property that they both make prior memory updates “visible” to all other processors in a shared-memory system. Hence, the following definition.

Definition 7 *A* revealing operation *makes updates to all shared variables performed in the current thread prior to the operation visible to other processors. I.e., a processor reading a shared variable immediately after the operation would obtain the same value as the revealing thread would immediately before the operation.*

Without a revealing operation, updates may never be propagated to other processors. The statement of our first theorem is not particularly surprising, but it is interesting to see where the requirement of a revealing operation arises in the proof.

Theorem 3 *Any mutual exclusion protocol that ensures freedom from starvation must use a revealing operation within every matched lock-unlock pair for each thread in the protocol.*

PROOF The proof is by contradiction. Suppose there is a protocol meeting the assumptions; i.e., it (C1) ensures mutual exclusion, (C2) ensures starvation freedom for each thread, assuming each thread stays in its critical section for a finite amount of time, and (C3) and does so assuming a demonic scheduler.

Consider the operation of the protocol on a pair of threads, A and B, where operations in A's lock, unlock, and critical section code do not use any revealing operation. Suppose that A and B start at their initial state. If A is at a lock operation and runs by itself, by (C2), it must enter its critical section. After the point where A enters its critical section, consider a new continuation, E1, where B executes its lock instruction. By (C1), thread B is enabled but must wait since A is in its critical section.

Continue E1 so that thread A exits its critical section and then B runs by itself. By (C2), B must enter its critical section. The decision by B to enter its critical section cannot be made on local information alone since otherwise there is a different schedule where, by (C3), the demonic scheduler can give sufficient time to B to make its decision while A is in its critical region, violating (C1). Thus, between the point in E1 where B waits to the point where it enters its critical section, A must have changed at least one global variable also visible to B and these critical changes must have been made be visible to B.

Now consider an alternative execution, E2, from the point in E1 where A enters its critical section and B is waiting such that in E2, thread A exits its critical section but changes to the global variables by A are not made visible to B. Such an execution is allowed since A is assumed not to execute a revealing operation within its lock-unlock actions. Without a revealing operation, the architecture is not constrained to make the shared-variable updates in A visible to B. Thus, the state visible to B is unchanged. There are two cases to consider at this point. If A cannot acquire the lock again (e.g., if the lock is turn based), then both A and B are blocked, leading to starvation. If A can acquire the lock, the prior sequence can be repeated, leading to starvation for B. In either case, thread B does not enter its critical section, contradicting (C2). □

This theorem raises the question of whether revealing operations can be eliminated by giving up starvation freedom. The next theorem shows that this is not possible for most standard protocols, all of which have the following property.

Definition 8 *A* symmetric choice *point in a mutual exclusion protocol is a state where two or more threads are waiting to enter a critical section and* either *thread can win the race by executing a sequence of its own actions.*

A mutual exclusion protocol has the *symmetric choice* property if there is a reachable symmetric choice point. The standard mutual exclusion protocols by Dekker, Peterson, and Lamport as well as the spin-lock protocol presented in the introduction have the symmetric choice property.

Theorem 4 *A mutual exclusion protocol requires a revealing operation for each acquire operation at a symmetric choice point.*

PROOF by contradiction. Suppose there is a mutual exclusion protocol with a symmetric choice point, *s*, where two threads, A and B are waiting to enter the critical section and A does not have a revealing operation in its acquire operation.

By definition of symmetric choice, there is an execution E1 from *s* where A acquires the lock first and another execution, E2, from *s* where B acquires the lock first. Construct execution E3 by first executing E1, then E2. Since there is no revealing operation in E1, the values of the shared variables as seen by process B at the end of E1 are the same as that in *s*, and the local state of *B* is unchanged by E1 (*B* remains in its waiting state). Therefore, it is possible to append execution E2 to E1 and get E1;E2, but the sequence E1;E2 results in both A and B acquiring the lock concurrently, violating mutual exclusion. □

13.5 Asymmetric Locks

Theorem 4 implies an algorithm that avoids revealing operations in locks must not have a symmetric choice state—i.e., it must be asymmetric. We now present such an algorithm.

For this section only, we return to assuming there is a fixed, known dominant thread. The algorithm is made asymmetric by forcing the nondominant threads to request permission from the dominant thread to proceed. Figure 13.5 shows the algorithm.

Before entering the critical section, the dominant thread checks whether another thread is accessing the critical section by probing the grant variable (line 10). While leaving the critical section (lines 20–24), it checks the request flag to determine whether another (nondominant) thread wishes to enter the critical section. If the flag is set, the dominant process hands the lock to the other thread by calling *fence* and setting the *grant* variable to 1. The call to *fence* commits any changes to shared variables made in the critical section before it passes the lock to any other thread.

A nondominant thread that desires to enter the critical section (lines 12–14) must first acquire a n-process lock, then set the request flag and wait for a grant. While leaving the critical section (lines 26–28), it resets the *grant* variable after calling *fence*. The call to *fence* commits all local changes to the main memory before the lock is passed back to the dominant process.

This method has the disadvantage that a nondominant requesting thread must wait for the dominant process to grant it permission. This implies the dominant thread must periodically check the request flag. Thus, the algorithm ensures starvation freedom for the nondominant threads only when the dominant thread checks the request flag infinitely often in any infinite computation. This can be ensured by periodically polling the request flag.

The advantage of the algorithm is that the dominant thread does not use a compare-and-swap instruction and uses a fence instruction *only* only when it passes control of the critical region to a nondominant thread. In periods of no contention from other threads, the dominant thread does not use any atomic or fence instructions, so locking incurs very little overhead.

13.6 Read-Write Biased Locks

In this section, instead of considering only exclusive locks, we discuss the design of biased read-write locks that incur very little overhead on the dominant thread. In general, a read-write lock allows either multiple readers or a single writer to access a critical section at any time.

We use a combination of a 2-process lock and a n-process lock. For the 2-process lock, we use a modified version of Peterson's algorithm; see Figure 13.6 and Figure 13.7. The flag variable can take three values: *READ*, *WRITE*, and *UNLOCK*. When a dominant thread *i* tries to obtain a read lock, it spins if at the same time there is another thread *j* writing (lines 13–16 in Figure 13.6). When the dominant thread tries to obtain a write lock, it waits if there is another thread that is either reading or writing (lines 4–7, Figure 13.7).

```
typedef struct {
  ThreadId owner;
  lockN n; /* N-process lock */
  bool request;
  bool grant;
} Lock;

biased_lock(Lock *l) {
  if (this_thread_id == l->owner)
    while (l->grant) {} /* wait */
  else {
    lockN(l->n);
    l->request = 1;
    while (!l->grant) {} /* wait */
  }
}

biased_unlock(Lock *l) {
  if (this_thread_id == l->owner) {
    if (l->request) {
      l->request = 0;
      fence(); /* make visible all memory updates */
      l->grant = 1; fence();
    }
  } else {
    fence();
    l->grant = 0;
    unlockN(l->n);
  }
}
```

Figure 13.5: Our asymmetric lock algorithm

For a nondominant process to acquire a write lock (lines 10–13, Figure 13.7), it first acquires a normal *n*-process write lock, *rwn*. This write lock *rwn* is contended only by nondominant processes. Once this lock is obtained, the process checks if the dominating process is in the unlock state and then enters the critical section. At this point the nondominant process is the only process in the critical section because the *rwlockN* provides exclusive access among the nondominant processes. The Peterson-like algorithm that follows it provides exclusive access from the dominant thread.

For a nondominant process to acquire a read lock (lines 18–28, Figure 13.6), it first acquires a normal *n*-process read lock on *n*. Since the *n*-process read lock on *rwn* can be held by multiple non-dominating processes, the first nondominant reader competes with the dominant process. If the dominant process is busy writ-

```
   typedef struct {
 2   ThreadId owner;
     int flagi; /* Owner's flag */
 4   int flagj; /* Non-owner's flag */
     bool turn;
 6   RWlockN rwn; /* N-process read-write lock */
     LockN n; /* N-process lock*/
 8   int non_owner_readers; /* No. of nondominant readers */
   } Lock;
10
   biased_r_lock(Lock *l) {
12   if (this_thread_id == l->owner) {
       l->flagi = READ;
14     l->turn = j;
       fence();
16     while (l->turn == j && l->flagj == WRITE) {}
     } else {
18     rwlockN(l->rwn, READ); /* Get a read lock */
       lockN(l->n); /* Get an exclusive lock */
20     l->non_owner_readers++;
       if (l->non_owner_readers == 1) {
22       /* First nondominant reader */
         l->flagj = READ;
24       l->turn = i;
         fence();
26       while (l->turn == i && l->flagi == WRITE) {}
       }
28     unlockN(l->n);
     }
30 }

32
   biased_r_unlock(Lock *l) {
34   if (this_thread_id == l->owner) {
       l->flagi = UNLOCK;
36     fence();
     }
38   else {
       lockN(l->n);
40     l->non_owner_readers--;
       if (l->non_owner_readers == 0)
42       l->flagj = UNLOCK;
       unlockN(l->n);
44     rwunlockN(l->rwn);
     }
46 }
```

Figure 13.6: Read functions of biased read-write locks

ing, the first nondominant reader spins on the flag variable. The last nondominant reader to exit the critical section sets the *flagj* variable to *UNLOCK*(lines 41–42, Figure 13.6). The first and last readers are maintained by a counter variable *non_owner_readers* and the field is protected by a normal *n*-process lock *n*.

As in the previous sections, for the dominant process to obtain either a read lock (lines 12–16 in Figure 13.6) or write lock (lines 3–7 in Figure 13.7) when there is no contention, requires the manipulation of only two flags, which results in far less overhead than normal *n*-process read-write locks.

The *rwlockN* function, which obtains a normal *n*-process read or write lock, can use standard reader-writer locks and implemented to be reader starvation-free or writer starvation-free. Between the dominant and nondominant process, the writer dominant process may starve, especially when nondominant readers keep coming

```
biased_w_lock(Lock *l)
{
  if (this_thread_id == owner) {
    l->flagi = WRITE;
    l->turn = j;
    fence();
    while (l->turn == j && l->flagj != UNLOCK) {}
  } else {
    rwlockN(l->rwn, WRITE);
    l->flagj = WRITE;
    l->turn = i;
    fence();
    while (l->turn == i && l->flagi != UNLOCK) {}
  }
}

biased_w_unlock(Lock *l) {
  if (this_thread_id == l->owner) {
    l->flagi = UNLOCK;
    fence();
  }
  else {
    l->flagj = UNLOCK;
    l->rwunlockN(l->rwn);
  }
}
```

Figure 13.7: Write functions of biased read-write locks

in and never relinquishing the lock. But since these readers are nondominant, we expect the readers to arrive infrequently. Therefore, starvation is unlikely in practice.

13.7 Algorithm Verification

The correctness of the algorithm presented in Section 13.2 can be inferred easily from its construction. The *n*-lock provides mutual exclusion among nondominant threads. The 2-lock provides mutual exclusion among the dominant and the nondominant thread.

The correctness of the asymmetric algorithm is less obvious and in fact, we discovered several pitfalls while developing it. We verified the algorithm from Section 13.5 using the SPIN [63] model checker. We created two processes, one dominant; the other nondominant, and verified mutual exclusion and progress properties. Even when there is more than one nondominant thread in the system, mutual exclusion holds because the normal *n*-lock provides exclusive access among the nondominant threads. The progress property also holds if the normal lock satisfies the progress property. The bounded waiting property however is not satisfied because nondominant threads are dependent on the dominant thread to acquire the lock.

For the ownership transfer protocol, we used SPIN to verify a configuration with one dominant thread and two nondominant threads. Each nondominant thread attempts nondeterministically to change ownership. We believe that this configuration describes all interesting interactions; the generalization of this automatic proof to arbitrary numbers of threads is ongoing work.

We also verified the biased read-write protocol using SPIN. We coded one dominant thread and two nondominant threads. Each one nondeterministically attempts a read or a write lock. The mutual exclusion property is satisfied even when there are more than two nondominant threads because the nondominant thread has to acquire either a normal n-write-lock or n-read-lock depending on the action before entering the critical section.

The verification with SPIN is based on a sequentially consistent model. By perturbing the sequence of assignments, it is possible to discover which orderings are relevant for the proof of correctness; this indicates positions where fences must be inserted for correctness on modern architectures with weaker ordering guarantees. In the future, we plan to use tools such as Check-Fence [22] to determine optimum placement of fences.

13.8 Experimental Results

The experiments described in this section have two purposes: to compare the performance of the new biased lock algorithms against similar algorithms proposed in earlier work using the pthread spin-lock implementation on Linux as the base reference, and to confirm our intuition about the behavior of these algorithms, i.e., that the performance improves monotonically with increasing domination. We coded the algorithms in C and we ran the experiments on an Intel Core 2 Quad processor with 2GB Memory and Fedora Core 7 installed. Programs were compiled with -O flag. We reimplemented reference [12] in C.

13.8.1 Performance with varying domination

To compare the different algorithms, we created four threads and made one of them dominant. The critical section just incremented a counter—a deliberately small task to maximize lock overhead. We varied the dominance percentage and measured the execution times; see Figure 13.8. A dominance of 90% indicates that for 100 accesses to the critical section, the dominant thread accesses the critical section 90 times and the remaining threads access the critical section 10 times. The lock accesses were evenly spaced: they follow a skewed but nonbursty access pattern.

We tested our micro benchmark on various algorithms. Our base case for comparison is the pthread spin lock (represented by a horizontal line at 0). We described the biased asymmetric lock in Section 13.5. The biased thread implementation uses Peterson's algorithm for 2-lock and p-threads for *n*-lock. The biased MCS implementation uses Peterson along with MCS locks. The biased CAS is the implementation from Kawachiya et al. [94]

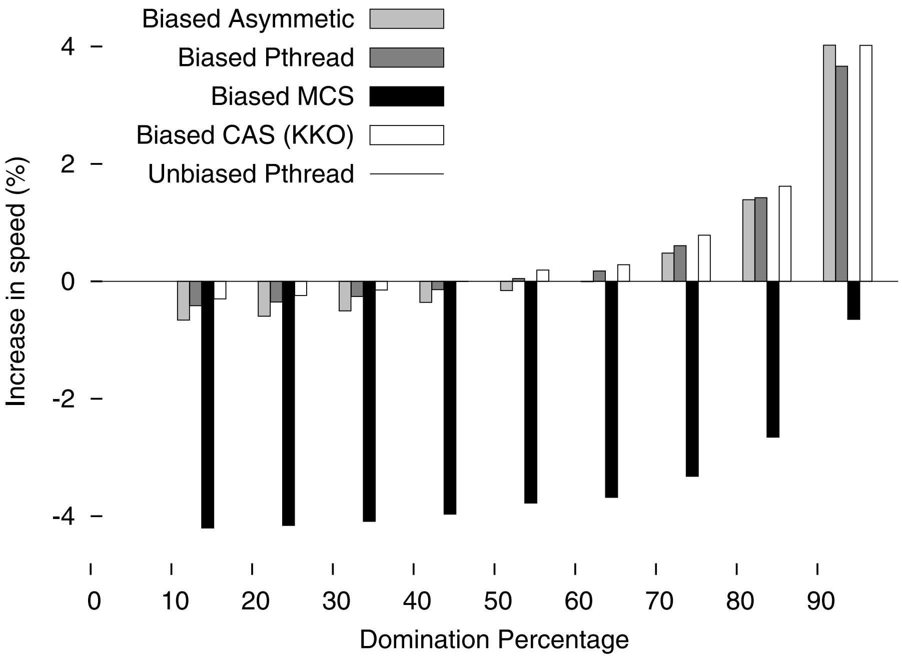


Figure 13.8: Behavior at varying domination percentages

For each of these algorithms, we observed the performance improve as we increased the dominance of the owner thread. Figure 13.9 shows details of the results from Figure 13.8 for domination between 90 and 100%, the expected range for the motivating packet processing application. Not surprisingly, the asymmetric method performs best when the domination percentage is high because asymmetric locks are very lightweight and do not require fence instructions in the dominant thread when there is no intervention from other threads. On the other hand, when the domination is less, the nondominant threads have to wait until the dominant thread signals; this overhead is insignificant for high dominance.

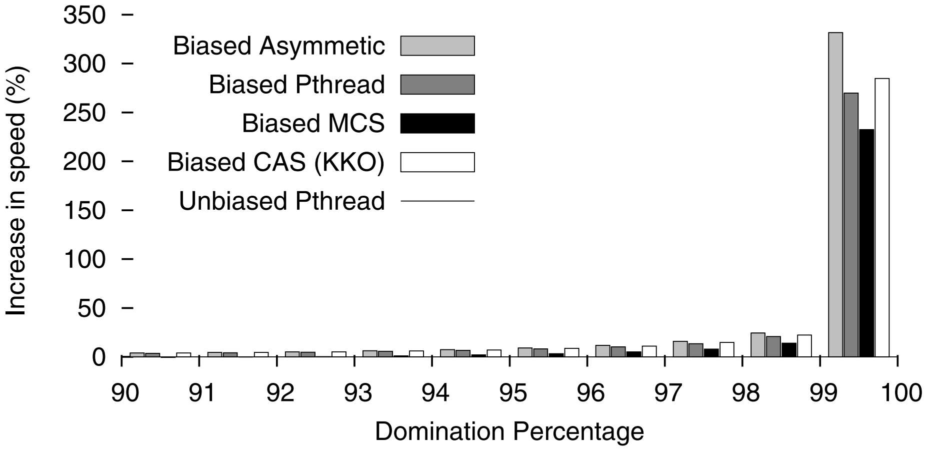


Figure 13.9: Behavior at high domination percentages

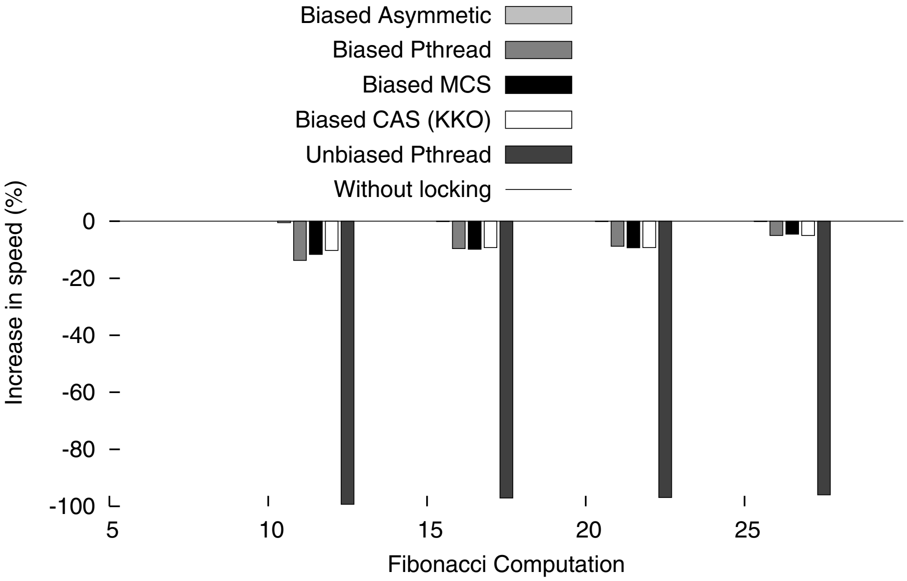


Figure 13.10: Lock overhead for a sequential program

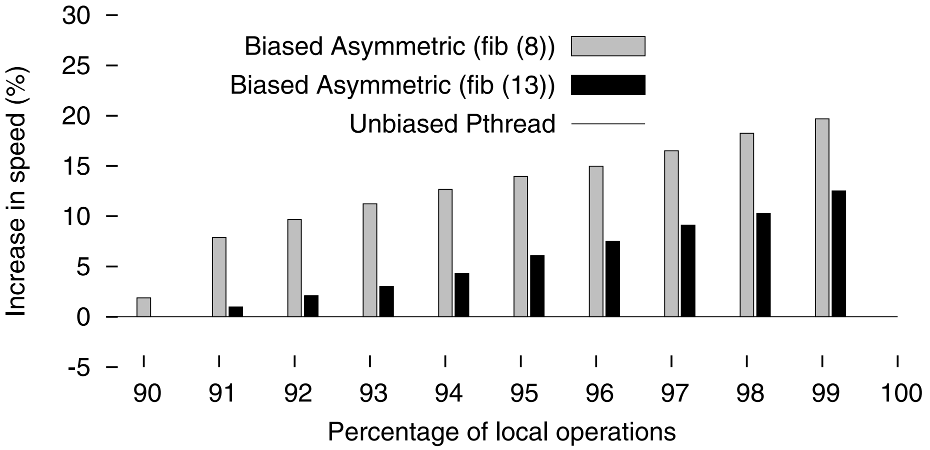


Figure 13.11: Behavior of our packet-processing simulator with asymmetric locks

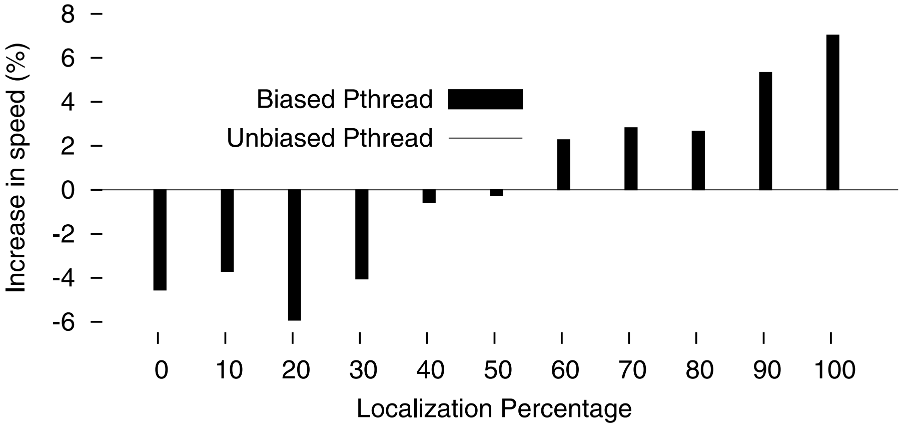


Figure 13.12: Performance of our biased locks on a database simulator for the query SELECT SUM(C2) GROUP BY C1

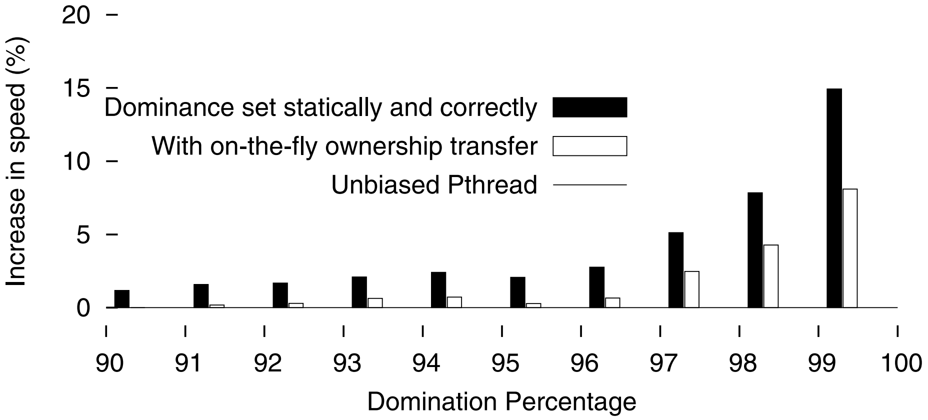


Figure 13.13: The effect of bias transfer

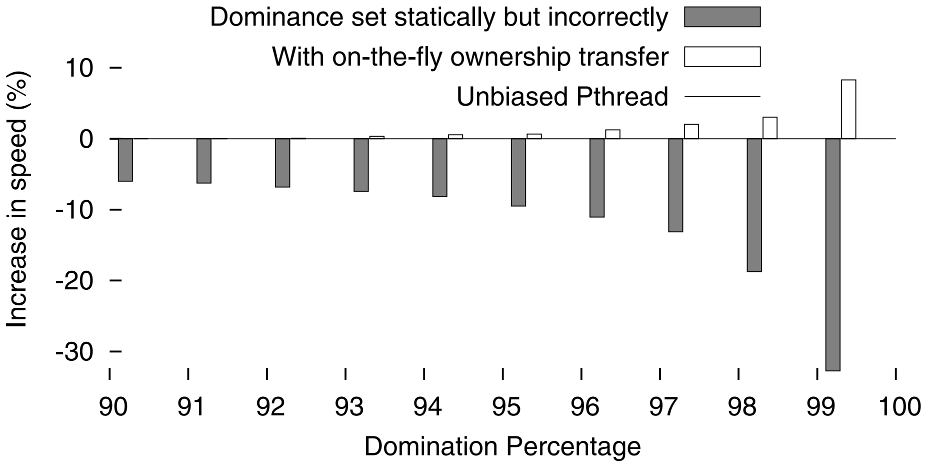


Figure 13.14: The effect of bias transfer for incorrect biasing

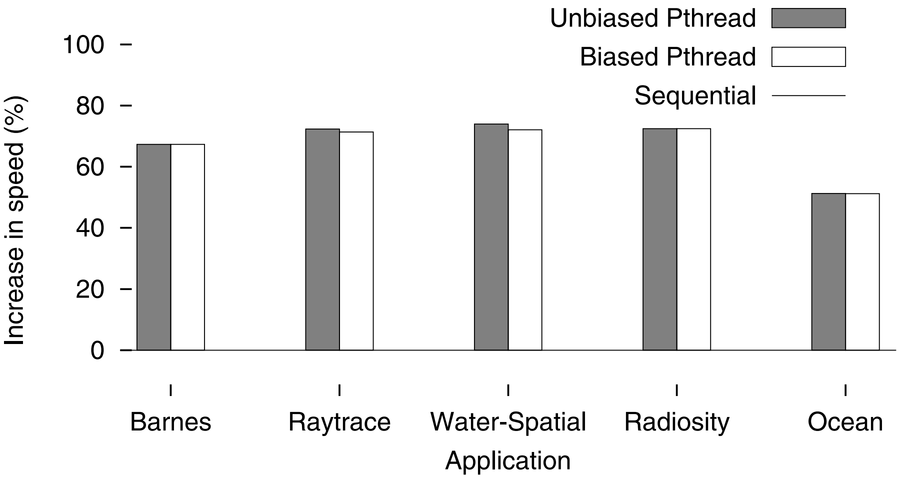


Figure 13.15: Performance of our biased locks on applications (SPLASH2 benchmark) without dominant behavior.

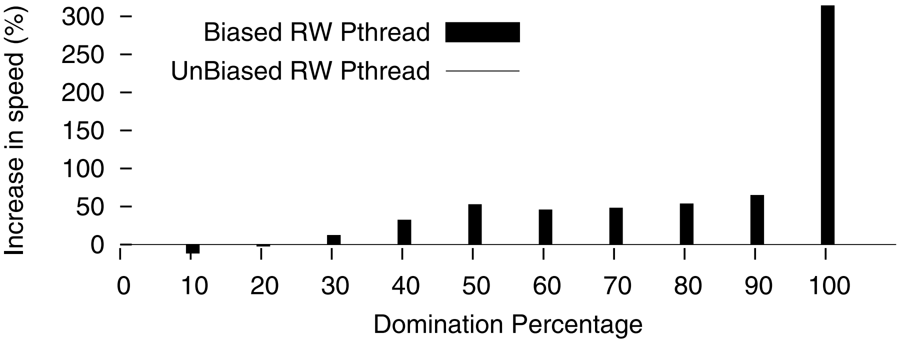


Figure 13.16: A comparison of our biased rwlock with Linux thread rwlock.

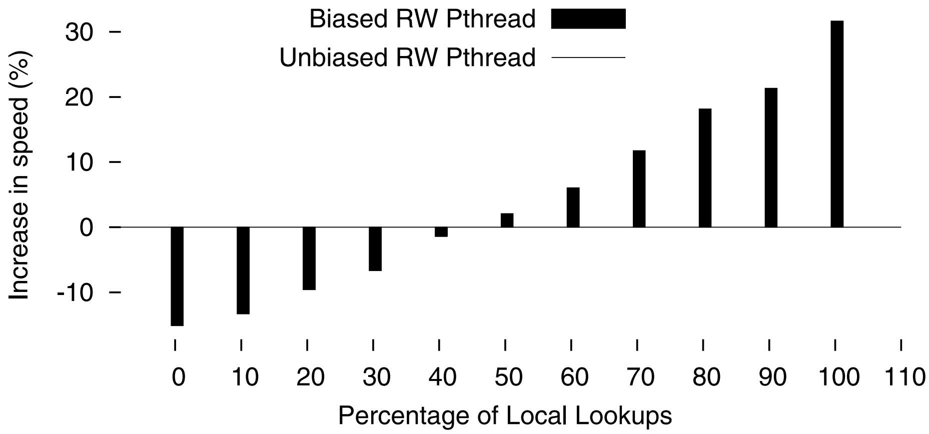


Figure 13.17: Performance of our biased read-write locks on a router simulator with 95% reads and 5% writes.

13.8.2 Locked vs. lockless sequential computation

Next, to measure the overhead of each of these locks, we created a sequential program that, to represent work, does the naïve recursive Fibonacci computation $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, and thus shows exponential behavior with increasing n. We protected the counter by a lock and compared the performance of different locks with the version without locks; see Figure 13.10. This setup merely measures the overhead of these locks. First, we see that the thread locks has the maximum overhead (about 100%) and asymmetric locks has the least (less than 1%). Second, as the computation load increases, the relative overhead decreases slowly.

13.8.3 Performance of a packet-processing simulator with asymmetric locks

From these experiments, we concluded that asymmetric locks are the best for our packet-processing application. Also, since the nondominant threads require permission from the dominant/owner thread to enter the critical section, asymmetric locks are suitable for applications that have dominant threads that run forever. In our packet-processing application, we replaced the thread locks by our asymmetric locks and compared the performance with the original one with pthreads (Figure 13.11). Within each lock we also added a synthetic computation that calculates Fibonacci numbers. When the computation time is high (e.g., for fib(13)), the nondominant threads have to wait more for the dominant thread to signal, therefore we see fib(8) performing better than fib(13). The difference between the two loads is roughly a factor of 10 because fib(n) scales as ϕ^n, where $\phi \approx 1.618$ is the golden ratio.

13.8.4 Biased Locks for Database Queries

Flexible locks (Section 13.2) that consist of 2-locks combined with n-locks are more robust to variations in computational load, although they require fence instructions whenever a dominant lock is obtained or released. To test the behavior of these locks, we wrote code that performs the SOL query "SELECT aggregate_function(c1) FROM t group by c2." Such a query is typically processed concurrently. The table t is divided into n parts; each part is processed by a separate thread, which maintains a local hash table. If the data c2 in t is localized, most of the hash updates are local to the thread, otherwise it is necessary to modify the hash table of a different thread; see Figure 13.12. As the locality of data increases, the biased locks perform better. Although the performance depends strongly on how the data are ordered, in many cases the ordering is such that data are localized.

13.8.5 Ownership transfer

We fixed the ownership of locks in the above applications, but our algorithm in Section 13.3 allows for ownership transfer. To test its performance, we created four threads that each perform a Fibonacci calculation in the critical section. Figure 13.13 compares the performance of our biased locks that supports on-the-fly ownership changes with the implementation that only supports static ownership. The implementation that supports change of ownership does not do as well as the the static implementation because of the extra overhead to support bias transfer. The ownership changes to the thread that was recently dominant, i.e, the most recent thread that has been acquiring the lock continuously. However, it does better than the unbiased implementation.

13.8.6 Ownership transfer with incorrect dominance

In Figure 13.14, we also compare the ownership on-the-fly implementation with a static ownership implementation, but for the latter implementation, we set the dominance incorrectly. The ownership on-the-fly implementation easily adapts itself and changes the dominance to the most recently dominant.

13.8.7 Overheads for nondominant behavior

The general trend is that as the dominance increases, biased locks perform better than unbiased locks. With applications that do not exhibit dominant behavior, we do not expect any improvement. We tested our biased locks on SPLASH2 benchmarks [134]. Most of these benchmarks exhibit master-slave behavior where work is divided among different threads. Even in the absence of dominance, our biased implementation deteriorated by at most 2% compared to the sequential version for these benchmarks.

13.8.8 Performance of biased read-write locks

Finally, we compared our biased read-write lock implementation with the pthread implementation of read-write locks. Figure 13.16 shows the results. Our biased read-write lock performs very well even when the dominance fraction is not very high because read-write locks are generally very expensive and our dominant read-write lock optimizes it to a large extent.

13.8.9 Performance on a simulated router application

To test the effect of biased read-write locks on actual examples, we simulated a router application. A router maintains a look-up table where the entries are mostly static, but occasionally (5% of the time) the IP addresses change, in which case a write lock is required. It usually maintains a distributed look-up table in which most lookups are local to a thread. Figure 13.17 suggests that, as expected, as we increase the number of local lookups, the biased read-write locks perform better.

13.9 Related Work and Conclusions

We have provided simple algorithms for constructing biased locks. We implemented these algorithms as a simple library, without any special support from the operating system. Our experimental evaluation shows that our algorithms perform well in practice when the dominance fraction is high, as expected. This matches the profile of our intended applications, e.g., network packet processing. The evaluations were all carried out on an Intel Quad core machine and the results, therefore, reflect the relatively high costs of fence and atomic operations on the x86 architecture.

As we mentioned in the introduction, there is other work on optimizing lock implementations, such as thin locks [8] and lock-reservation algorithms [71; 94; 7]. The original thin lock algorithm requires a compare-and-swap on each lock acquisition, which our algorithm avoids.

The lock-reservation work is closest to ours. In Kawachiya et al. [71], the disadvantage is that when a lock is reserved for the owner and the nonowner tries to attempt the lock, the nonowner stops the owner thread and replaces a lock word. This step is very expensive because the owner thread is suspended. Onodera et al. [94] proposes a modification similar to ours: a hybrid algorithm that tightly intertwines Dekker's 2-process algorithm with an n-process CAS algorithm. Our scheme simplifies this by keeping the two algorithms separate and generalizes it by allowing any choice of 2-process and n-process mutual algorithms.

We show how to transfer lock ownership among threads without suspending the current owner. Although Russell and Detlefs [104] also support bias transfer, their global safe-point technique for bias revocation is costly. In their technique, it is difficult to determine at any point whether a biased lock is actually held by a given thread. Our technique is simple and inexpensive: it only requires two extra assignments and two comparisons.

Finally, we examined the necessity of memory fence instructions on modern processors and shed light on the key role played by the symmetric choice property of most mutual-exclusion algorithms. The asymmetric algorithm presented in Section 13.5 is, in a sense, the most efficient possible, since it avoids both memory fence and atomic operations in the dominant process except at the point of transferring control of the lock, where they are unavoidable. Earlier work on asymmetric biased locks [43; 44] has a similar motivation, but the analogues of the request-grant protocol, called SERIALIZE(t) by Dice et al. [43], appear fairly heavyweight, involving either thread suspension and program counter examination, or context switches.

It is clear that the performance improvement of biased locks depends on the relative performance of compare-and-swap, memory fence, and simple memory instructions. There is unfortunately no standard model that one can use to theoretically analyze performance, therefore we picked the instance of the most common architecture for our experiments. We plan to experiment on more machines in the future.

Lastly, we have not yet used biased locks in our SHIM models and compilers that guarantee determinism, even though their deterministic constructs are implemented using locks. We wish to do this soon.

Part V

Conclusions

Chapter 14

Conclusions

It is time for a new era of bug-free parallel programming that will enable programmers to shift easily from the sequential to the parallel world. I believe that this thesis will be a significant step along the way to parallel programming.

This thesis provides programming language support to address the two major problems of concurrency — nondeterminism and deadlocks. Through this thesis, we demonstrate that determinism is not a huge performance bottleneck. It has advantages for code synthesis, optimization, and verification, making it easier for an automated tool to understand a program's behavior. This advantage is particularly helpful for deadlock detection, which for models like SHIM can ignore differently interleaved executions.

Most concurrent programming languages that are in use today allow programmers to write programs that are nondeterministic and/or prone to deadlocks. These bugs are usually checked during runtime. This thesis presents a way to avoid these bugs during the software development phase.

We believe that our techniques simplify debugging and hence enhance the productivity of programmers. The language and the compiler simply prevent nondeterminism and deadlocks. The programmer does not have to worry about these concurrency bugs and can focus on the logic of the program.

Since we adopt our techniques at the compiler level, the application-level programmer does not have to worry about the hardware. This enhances portability, especially when the underlying hardware changes. Also, we also do not propose changes in the hardware; a software amendment is always easier and faster to apply.

Although we have presented our techniques with SHIM, we believe that our ideas can be extended to any general programming language. In Chapter 10, we adopted our model to the X10 programming language. Realistically, it will take one engineer and approximately six months to port our ideas to any general-purpose concurrent programming language. Some of the techniques we have discussed are already in the official X10 release. A good way to evaluate the practical use of this thesis is to adopt these ideas in many other concurrent programming languages. We would like to see how their benchmarks perform and if there is significant performance overhead. In addition, we would like to see how programmer productivity increases.

Our future plans for SHIM include code generation fusing parallelism with static scheduling [48], extracting parallelism [82], data distribution [113], communication optimization [102], synthesis of hardware and dealing with reactive systems.

We currently do not deal with pointers and complicated data structures, and we need a good mechanism to include them. Our long term goal is automatic determinism [125]— starting from a nondeterministic program, our compiler will insert just enough additional synchronization to guarantee deterministic behavior, even in the presence of nondeterministic scheduling choices. Our ultimate goal is deterministic deadlock-free concurrency along with efficiency.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 183–193, New York, NY, USA, 2007. ACM.

[3] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, second edition, 2006.

[4] Thomas William Ainsworth and Timothy Mark Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, September 2007.

[5] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2009. ACM.

[6] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of Operating System Design and Implemetation (OSDI)*, pages 193–206, Vancouver, BC, Canada, October 2010.

[7] David Bacon and Stephen Fink. Method and apparatus to provide concurrency control over objects without atomic operations on non-shared objects. US Patent Docket No. YO999-614, 2000.

[8] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

[9] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 382–400, Minneapolis, Minnesota, October 2000.

[10] Howard Barringer, Dimitra Giannakopoulou, , and Corina S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proceedings of the 2nd International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 14–21, Helsinki, Finland, 2003. Iowa State University Technical Report #03–11.

[11] Yosi Ben-Asher, Eitan Farchi, and Yaniv Eytani. Heuristics for finding concurrent bugs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 288, Nice, France, April 2003.

[12] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *Lecture Notes in Computer Science*, pages 64–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 41–50, Portland, Maine, July 2006.

[14] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In *Compositionality: The Significant Difference (COMPOS)*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102, Bad Malente, Germany, September 1998.

[15] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[16] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, New York, NY, USA, 2009. ACM.

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

[18] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311, Glasgow, UK, July 2005.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[20] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2009. ACM.

[21] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[22] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 12–21, San Diego, California, USA, June 2007.

[23] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 3–12, New York, NY, USA, 2009. ACM.

[24] Sagar Chaki, Edmund Clarke, Joël Ouaknine, and Natasha Sharygina. Automated, compositional and iterative deadlock detection. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, California, June 2004.

[25] Sagar Chaki, Joël Ouaknine, Karen Yorav, and Edmund Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. *Electronic Notes in Theoretical Computer Science*, 89(3):417–432, 2003.

[26] Sagar Chaki and Nishant Sinha. Assume-guarantee reasoning for deadlock. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 134–141, San Jose, California, November 2006.

[27] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 11–22, New York, NY, USA, 2008. ACM.

[28] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983.

[29] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[30] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[31] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM.

[32] Alex Chunghen Chow, Gordon C. Fossum, and Daniel A. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. In *Global Signal Processing Expo (GSPx)*, Santa Clara, California, October 2005. Available from IBM.

[33] Marek Chrobak, János Csirik, Csanád Imreh, John Noga, Jiri Sgall, and Gerhard J. Woeginger. The buffer minimization problem for multiprocessor scheduling with conflicts. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of

Lecture Notes in Computer Science, pages 862–874, Heraklion, Crete, Greece, 2001. Springer-Verlag.

[34] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An OpenSource tool for symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, Copenhagen, Denmark, July 2002.

[35] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS)*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[36] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[37] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Chicago, Illinois, July 2000.

[38] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering Methodology*, 17(2):1–52, 2008.

[39] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2003.

[40] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.

[41] Eddy de Greef, Francky Catthoor, and Hugo de Man. Array placement for storage size reduction in embedded multimedia systems. In *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific*

Systems, Architectures and Processors, page 66, Washington, DC, USA, 1997. IEEE Computer Society.

[42] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96. ACM, 2009.

[43] Dave Dice, Hui Huang, and Mingyao Yang. Asymmetric dekker synchronization. Technical report, Sun Microsystems, 2001. http://home.comcast.net/ pjbishop/Dave.

[44] Dave Dice, Mark Moir, and William Scherer. Quickly reacquirable locks. Technical report, Sun Microsystems, 2003. http://home.comcast.net/ pjbishop/Dave.

[45] Edsger W. Dijkstra. Cooperating sequential processes. *Technological University, Eindhoven, The Netherlands, September 1965. Reprinted in Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, 43-112*, 1965.

[46] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock free data structures using STM in Haskell. In *Proceedings of Functional and Logic Programming (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*, pages 65–80, Fuji Susono, Japan, April 2006.

[47] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[48] Stephen A. Edwards and Olivier Tardieu. Efficient code generation from SHIM models. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 125–134, Ottawa, Canada, June 2006.

[49] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, August 2006.

[50] Stephen A. Edwards and Nalini Vasudevan. Compiling SHIM. In Sandeep K. Shukla and Jean-Pierre Talpin, editors, *Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction*, chapter 4, pages 121–146. Springer, 2010.

[51] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, March 2008.

[52] Stephen A. Edwards and Jia Zeng. Static elaboration of recursion for concurrent software. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 71–80, San Francisco, California, January 2008.

[53] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[54] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin K. O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the CELL processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 161–172, Saint Louis, Missouri, September 2005.

[55] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, Tampa, Florida, 2006. Article 83.

[56] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High performance sorting on the Cell processor. In *Proceedings of Very Large Data Bases (VLDB)*, pages 1286–1297, Vienna, Austria, September 2007.

[57] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Design Automation Conference*, pages 819–824, Anaheim, California, 2005. ACM.

[58] R. Govindarajan, Guang R. Gao, and Palash Desai Y. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing Systems*, 31(3):207–209, July 2002.

[59] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, Chicago, Illinois, USA, June 2005.

[60] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.

[61] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[62] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[63] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[64] IBM. *Cell Broadband Engine Architecture*, October 2007. Version 1.02.

[65] IBM. *Example Library API Reference*, September 2007. Version 3.0.

[66] IBM et al. T. J. Watson libraries for analysis, 2006. http://wala.sourceforge.net.

[67] Intel IA-32 Architecture Software Developer's Manual, vol. 3A, 2009.

[68] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of Principles of Programming Languages (POPL)*, pages 295–308, St. Petersburg Beach, Florida, January 1996.

[69] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[70] 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.

[71] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented*

programming, systems, languages, and applications, pages 130–141, New York, NY, USA, 2002. ACM.

[72] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May-June 2006.

[73] Jean-Pierre Krimm and Laurent Mounier. Compositional state space generation with partial order reductions for asynchronous communicating systems. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 266–282. Springer, 2000.

[74] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[75] Dongmyun Lee and Myunghwan Kim. A distributed scheme for dynamic deadlock detection and resolution. *Information Sciences*, 64(1–2):149–164, 1992.

[76] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[77] Flavio Lerda, Nishant Sinha, and Michael Theobald. Symbolic model checking of software. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[78] Bill Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 211–217, Paris, France, February 1998.

[79] E. Malaguti. The vertex coloring problem and its generalizations. *4OR: A Quarterly Journal of Operations Research*, 7:101–104, March 2009.

[80] E. Malaguti, M. Monaci, and P. Toth. Models and heuristic algorithms for a weighted vertex coloring problem. *Journal of Heuristics*, 2008. DOI: 10.1007/s10732-008-9075-1.

[81] Stephen P. Masticola and Barbara G. Ryder. A model of Ada programs for static deadlock detection in polynomial times. In *Proceedings of the Workshop on Parallel and Distributed Debugging (PADD)*, pages 97–107, New York, NY, USA, May 1991. ACM.

[82] Sjoerd Meijer, Bart Kienhuis, Alex Turjan, and Erwin de Kock. A process splitting transformation for Kahn process networks. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1355–1360, Nice, France, April 2007.

[83] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[84] Nicolas Mercouroff. An algorithm for analyzing communicating processes. In *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, pages 312–325, London, UK, 1992. Springer.

[85] The Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Version 1.1.

[86] Praveen K. Murthy and Shuvra S. Bhattacharyya. Systematic consolidation of input and output buffers in synchronous dataflow specifications. *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 673–682, 2000.

[87] Praveen K. Murthy and Shuvra S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177–198, February 2001.

[88] Praveen K. Murthy and Shuvra S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 9(2):212–237, April 2004.

[89] Praveen K. Murthy and Shuvra S. Bhattacharyya. *Memory Management for Synthesis of DSP Software*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[90] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Journal of Formal Methods in System Design*, 32(3):207–234, June 2008.

[91] M. Ohara, H. Inoue, Y.Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, 2006.

[92] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 235–250, 2006.

[93] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, New York, NY, USA, 2009. ACM.

[94] Tamiya Onodera, Kiyokuni Kawachiya, and Akira Koseki. Lock reservation for Java reconsidered. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 559–583, 2004.

[95] OpenMP Architecture Review Board, www.openmp.org. *OpenMP C and C++ Application Program Interface*, March 2002. Version 2.0.

[96] G. L. Peterson. Myths about the mutual exclusion problem. *IPL 12(3)*, pages 115–116, 1981.

[97] Fabrizio Petrini, Gordon Fossum, Juan Fernández, Ana Lucia Varbanescu, Mike Kistler, and Michael Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, Long Beach, California, USA, March 2007.

[98] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, volume 1, pages 184–185, 582, San Francisco, California, February 2005.

[99] Jean pierre Krimm and Laurent Mounier. Compositional state space generation from Lotos programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1217 of *Lecture Notes in Computer Science*, pages 239–258, Enschende, The Netherlands, April 1997. Springer.

[100] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Proceedings of the NATO Advanced Study Institute on Logics*

and Models of Concurrent Systems, pages 123–144, La Colle-sur-Loup, France, 1985. Springer.

[101] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[102] John Reppy and Yingqi Xiao. Specialization of CML message-passing primitives. *SIGPLAN Notices*, 42(1):315–326, 2007.

[103] Paul Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, February 1991.

[104] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not.*, 41(10):263–272, 2006.

[105] Tarik Saidani, Stéphane Piskorski, Lionel Lacassagne, and Samir Bouaziz. Parallelization schemes for memory optimization on the Cell processor: A case study of image processing algorithm. In *Proceedings of the Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA)*, pages 9–16, Brastov, Romania, September 2007.

[106] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–271, New York, NY, USA, 2007. ACM.

[107] Enno Scholz. Four concurrency primitives for Haskell. In *ACM/IFIP Haskell Workshop*, pages 1–12, La Jolla, California, June 1995. Yale Research Report YALE/DCS/RR–1075.

[108] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 115–126, New York, NY, USA, 2005. ACM.

[109] Baolin Shao, Nalini Vasudevan, and Stephen A. Edwards. Compositional deadlock detection for rendezvous communication. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 59–66, Grenoble, France, October 2009.

[110] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[111] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, New York, NY, USA, 2001. ACM.

[112] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 21–33, New York, NY, USA, 2006. ACM.

[113] Philip Stanley-Marbell, Kanishka Lahiri, and Anand Raghunathan. Adaptive data placement in an embedded multiprocessor thread library. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 698–699, Munich, Germany, March 2006.

[114] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

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

[116] 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.

[117] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. Technical Report CUCS–036–06, Columbia University, Department of Computer Science, New York, New York, USA, September 2006.

[118] Jürgen Teich, Eckart Zitzler, and Shuvra S. Bhattacharyya. Buffer memory optimization in DSP applications—an evolutionary approach. In *Proceedings of Parallel Problem Solving from Nature (PPSN)*, volume 1498 of *Lecture Notes in Computer Science*, pages 885–896, Amsterdam, The Netherlands, September 1998. Springer-Verlag.

[119] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[120] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196, Grenoble, France, April 2002.

[121] Xinmin Tian, Milind Girkar, Aart Bik, and Hideki Saito. Practical compiler techniques on efficient multithreaded code generation for OpenMP programs. *The Computer Journal*, 48(5):588–601, 2005.

[122] Nalini Vasudevan and Stephen A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, Anaheim, California, June 2008.

[123] Nalini Vasudevan and Stephen A. Edwards. Buffer sharing in CSP-like programs. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Cambridge, Massachusetts, July 2009.

[124] Nalini Vasudevan and Stephen A. Edwards. Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore. In *Proceedings of the Symposium on Applied Computing (SAC)*, volume III, pages 1626–1631, Honolulu, Hawaii, March 2009.

[125] Nalini Vasudevan and Stephen A. Edwards. A determinizing compiler. In *Programming Languages Design and Implementation (PLDI) - Fun Ideas and Thoughts Session*, Dublin, Ireland, June 2009.

[126] Nalini Vasudevan and Stephen A. Edwards. Buffer sharing in rendezvous programs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(10):1471–1480, October 2010.

[127] Nalini Vasudevan and Stephen A. Edwards. Determinism should ensure deadlock-freedom. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, California, June 2010.

[128] Nalini Vasudevan, Stephen A. Edwards, Julian Dolby, and Vijay Saraswat. d^2c: Deterministic, deadlock-free concurrency. In *High Performance Computing - Student Research Symposium*, December 2010.

[129] Nalini Vasudevan, Kedar Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 65–74, Vienna, Austria, September 2010.

[130] Nalini Vasudevan, Satnam Singh, and Stephen A. Edwards. A deterministic multi-way rendezvous library for Haskell. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, Miami, Florida, April 2008.

[131] Nalini Vasudevan, Olivier Tardieu, Julian Dolby, and Stephen A. Edwards. Compile-time analysis and specialization of clocks in concurrent programs. In *Proceedings of Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*, pages 48–62, York, United Kingdom, March 2009.

[132] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 455–471. Springer Berlin / Heidelberg, 2011.

[133] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[134] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[135] Hao Zheng, Jared Ahrens, and Tian Xia. A compositional method with failure-preserving abstractions for asyn chronous design verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1343–1347, July 2008.