PROFILING INFRASTRUCTURE FOR THE PERFORMANCE EVALUATION OF ASYNCHRONOUS SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

David C. Fang

August 2008

PROFILING INFRASTRUCTURE FOR THE PERFORMANCE EVALUATION OF ASYNCHRONOUS SYSTEMS

David C. Fang, Ph.D.

Cornell University 2008

Designing and optimizing large-scale, asynchronous circuits is often an iterative process that cycles through synthesis, simulating, benchmarking, and program rewriting. Asynchronous circuits are usually specified by high-level, sequential or concurrent programs that prescribe the intended behavior. The self-timed nature of the interface gives designers much freedom to refine and rewrite equivalent specifications for improved circuit synthesis. However, at any step in the design cycle, one faces an uncountable number of choices for program rewriting — one simply cannot afford to explore all possible transformations. Informed optimizations and design space pruning can require detailed knowledge of the run-time behavior of the program, which is what our simulation trace analysis infrastructure provides. Tracing entire simulations gives users the opportunity to understand program execution in great detail. Most importantly, trace profiling captures typical run-time behavior and input-dependent behavior that cannot always be inferred from static analysis. Profiling provides valuable feedback for optimizing both high-level transformations and low-level netlist synthesis.

To address this need for profiling, we present a framework for analyzing the simulated execution of high-level, concurrent programs, as a foundation for iterative optimization and synthesis of asynchronous circuits. The framework includes a Scheme environment and a library of primitive procedures for handling and querying trace data. Interactivity is essential for analysis sessions where the sequence

of queries to execute is not known *a priori*. The initial library also includes procedures for some frequently run analyses (built on top of the primitives). Providing an interface for working directly with the simulation and trace data structures makes analysis development within our framework both flexible and convenient. The extensibility of our framework enables compilation-free development and prototyping of custom analysis routines, so users can easily share and build upon the work of others. The primary purpose of this analysis framework is to enable future tools to use profile-driven feedback in automating iterative optimization and design-space exploration.

BIOGRAPHICAL SKETCH

David is the son of Que-Tsang and Lee Fang, who emigrated from Taiwan as graduate students at the University of Illinois, Urbana-Champaign. He graduated from Franklin Regional Senior High School in Murrysville, PA with Honors with Highest Distinction in the class of 1997. He enrolled at the California Institute of Technology in 1997, ambitiously intending to tackle electrical engineering, physics, and a twist of applied mathematics, but graduated with only a Bachelor of Science in Electrical Engineering with Honors in 2001. He thanks his wise upperclassmen and peers at Caltech for dissuading him from attempting more than one major. David still reminisces about the times spent growing and learning at Caltech, that is, when he wasn't slaving away on projects and courses.

The roots of his interest in asynchronous VLSI trace back to the EE/CS181abc class he took as an undergraduate, taught by Prof. Alain Martin and his research group members. Once he grasped the beauty and purity of self-timed programming, he endeavored to make his contribution to the field. The author received a National Defense Science and Engineering Graduate Fellowship, sponsored by the Office of Naval Research. Since the summer of 2001, the author has been a student of the Computer Systems Laboratory in the Cornell Electrical and Computer Engineering Department, under the guidance of Prof. Rajit Manohar, a former student of Alain Martin. The change of climate was likened to leaping out of the frying pan and into the freezer.

To supplement his background in asynchronous VLSI, the author minored in computer science, and maintains interest in computer architecture and compilers, which spans the hardware and software aspects of computer engineering. He also maintains an interest in algorithms, numerical analysis, optimization problems, information theory, and mathematical puzzles. Aside from being passionate about

his work, he also maintains strong interest in music and dance, which have deserved a lot more time than he has actually devoted.

David has already accepted a full-time job as an engineer at Achronix Semiconductor Corporation in San Jose, CA, with many of his current and former Cornell colleagues, where he aspires to help take asynchronous circuits to new limits.

David hates writing about himself in the third person, so rest assured that the remainder of this dissertation is not written in this awkward manner.

dedicated to my grandparents,

who have always held the highest expectations of me

ACKNOWLEDGMENTS

Graduate school is not *just* about cranking out research papers and earning another degree. Much of my education in graduate school was gained through discussions with talented and uniquely-minded colleagues. The most valuable times were spent socializing outside of work. You have all made my stay at Cornell a truly memorable experience. I would like to extend my sincerest gratitude (or give a shout out) to the following individuals and organizations.

Mainak, I didn't make our national Math Olympiad team either – thank you for sharing your passion for solving challenging problems. Daehyun, your passion with video games is contagious! Avneesh, the Source will always be with you. Prof. Mark Heinrich, Prof. Evan Speight, the early years of CSL were entertaining, thanks to your presence in the faculty! Ilya, don't say I didn't warn you about Compilers. Brian, you're patience with school gives hope to many students. Jonathan, your rants about wine and women were a welcome respite from the daily grind of research. Scott, graduate student manager and savior by day, maniacal drummer by night – we know who holds the *real* power!

Teifel, you were right about grad school, not that I had any doubts. Clint, no only do I have you to thank for my job, but my days in the office just would've lacked amusement without your wit and humor, even when the joke was on me. Virantha, thank you for introducing me to `awk` many years ago, choose carefully next time. Song, there's no shame in being the Beast-Master. Sandra and Brett, thank you for sharing your passion for cats, computers, and food. Biermann, the legend of the Drunken-Duck Style Master lives on. LaFrieda, you can count on me for a football match anytime. Paula and Basit, thank you for bringing life to our memorable social gatherings. Rob and Ben, your days of glory are just over the horizon. Filipp, your destiny awaits you even when you deny it, search

your feelings, you know it to be true. Carlos Tadeo, your humility only earns you greater respect.

Prof. Martin Burtscher and Prof. Radu Rugina, thank you for serving on my committee during our years at Cornell; your feedback was always appreciated. Thank you, Prof. José Martínez and Prof. Dave Albonesi, for serving on my committee at the late hour, and especially for your constructive feedback and thoughtful questions. Prof. Rajit Manohar, thank you for your continued guidance, and always leading by good example for students and future educators. Your patience with me knows no limits, but I won't push it any longer!

Had it not been for my friends in Cornell Ballroom and Dancesport and the Ithaca Swing Dance Network, I would've gone crazy years ago. Thank you for the wonderful times and giving me an escape from work. These are the times I will miss the most in Ithaca. André, thank you for showing me that the road to success doesn't require long hours of work. Nate, Adriana, and Peter, thanks for tolerating my extended absences and odd hours at our humble apartment. Ellan, Sara, Heather, thank you for sharing many dances, travels, music, and hours of deep conversation over the years. Helen, your support and encouragement have guided me through some of the difficult times in school, thank you for reinforcing my determination. Mom, Dad, Allison – thank you for bearing with me and my long absences from home, in the pursuit of higher education.

Finally much of my work in our research group is funded by: Department of Defense, American Society of Engineering Education, Office of Naval Research, Defense Advanced Research Projects Agency, and National Science Foundation. Their generous support continues to attract talent to undertake advanced research at institutes of higher education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF PROGRAMS

LIST OF ABBREVIATIONS

AFPGA	asynchronous FPGA
API	application program interface
ASIC	application-specific integrated circuit
AVLSI	asynchronous VLSI
CAD	computer-aided design/development
CFG	control flow graph
CHP	Concurrent Hardware Processes
CSP	Communicating Sequential Processes
EDA	electronic design automation
FPGA	field-programmable gate array
GNU	GNU is not Unix
HAC	Hierarchical Asynchronous Circuits (HAC)
HDL	hardware description language
HSE	handshaking expansion
PRS	production rule set
QDI	quasi-delay insensitive
RTL	register transfer logic
SDT	syntax-directed translation
STA	static timing analysis
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	very large scale integration
WYWIWYG	what you write is what you get

PREFACE

A long, long time ago, in a university not so far away... a battle between synchronous and asynchronous circuit designers raged. The synchronous empire sought to maintain its stronghold over the semiconductor industry as the only way to design large-scale integrated digital systems. However, a band of rebel asynchronous designers have been holding out at a secret base, evading authorities, and plotting to launch an assault on the empire.

Part of the divide between the two parties is attributed to commercial design tool vendors' unwillingness and inability to support the rebel cause. They see no profit in supporting the rebels, as the rebels cannot muster enough demand for special arms (asynchronous tools). And yet much of the rebels disadvantage remains due to their humbler arsenal of weapons. The rebel struggle is often seen as an insurmountable uphill battle. Clearly, advancement of the rebels' technology must come from within.

This is an over-dramatization of the struggle of asynchronous VLSI, however, this prevailing sentiment is captured by the opening quote in the Introduction. Development of asynchronous design tools can only come from *within* the asynchronous academic community, those who understand its principles.

This dissertation addresses a question of how one goes about designing and optimizing asynchronous circuits. Numerous papers in the literature describe methods for mathematically translating sequential programs to parallel programs, and parallel programs to circuits. But does the theory reflect how circuits are designed in practice? (In theory, yes, but in practice, no.) The problem is simply that there are *too many* ways of accomplishing the same task, each with its own merits and tradeoffs. There is not only one translation, but many translations possible in synthesizing asynchronous circuits from high-level programs.

What navigates asynchronous designers through the sea of design choices? To some extent it is the limited set of instruments and fundamental principles that exist, but to this day, many large-scale custom asynchronous designs are guided by *experience*, wisdom accumulated from many journeys at sea. Experience comes from past designs and lessons learned with their successes and failures. However, if experience were easy to organize and *express*, there would be more literature on the trials and tribulations of circuit designers (not found in textbooks). (Sure, I'll accept that they might not be exciting to read, and in low demand.) Experience has been difficult to pass on to new engineers.

Future design tools beckon for a way of expressing knowledge beyond the fundamentals, so that the wisdom of the ancient masters would not be lost and unnecessarily re-learned with each passing generation. The expert systems area of artificial intelligence is one such approach where a large knowledge base coupled with an inference engine seeks to solve problems by interactively querying a user. At the crossroads of choosing my dissertation topic, I was considering several aspects of asynchronous circuit design from the high-level concurrent programs to the low-level circuit details. Synthesis, the lowering of abstraction from high-level programs to circuits, looked like a path well-trodden by my predecessors. Optimizations within low-level netlists, seemed to have less potential for improvement than high-level program rewriting and restructuring. To take an analogy from software development: a *structurally* optimized program can be synthesized (compiled) to potentially better circuits (or machine code).

However, the problem with attempting to statically analyze and restructure programs (like source-to-source compilation), is that the number of transformations available at any step is *unbounded*, and that the benefits of transformations were often non-obvious, input-dependent, or involved some tradeoff. Rather than spend

effort on program transformations themselves, another useful contribution would be some infrastructure for analyzing the merits of transformations, as guided by the experience of our predecessors. The ultimate goal was an infrastructure for analyses that would easily extend, as engineers contributed their knowledge. If the seas were marked with more lighthouses and warning beacons, navigation for future pioneers would be much easier. Thus, the work within this dissertation describes a method by which experienced designers can teach their apprentices to spot lights in the horizon and shallow rocks beneath the waters, and a means for apprentices to summon the foresight of those who have sailed before them.

David Fang

fang@csl.cornell.edu

CHAPTER 1

INTRODUCTION

> *I don't think it is a technical issue, but an infrastructure support problem. It's the chicken-or-egg question all over again: we cannot easily design asynchronous systems because appropriate tools aren't available. And there are no tools, the EDA houses say, because there is no demand for them.*
>
> Bernard Cole, August 2002 [9]

Many advantages of asynchronous circuits over their synchronous counterparts have been cited for years: robustness to delay variation, design scalability through modularity, formal verification from concurrent programs, energy efficiency due to event-driven activity in lieu of clocks. Despite these advantages, the absence of the asynchronous VLSI design methodology[1] from mainstream adoption has been largely attributed to the lack of design tools[2]. This dissertation is thus motivated by the ever-growing need for asynchronous VLSI design tools.

This dissertation corroborates the importance of profiling and analyzing program executions in choosing concurrent program transformations for optimization. The behavior of asynchronous circuits is specified using high-level concurrent *programs*; asynchronous circuits are implementations of concurrent programs. While there exist many methods for synthesizing circuits from programs, optimizations in concurrent programs translate to structurally optimized circuits. The work described herein is a powerful trace analysis framework for aiding the rewriting and refinement (transformation and optimization) of high-level asynchronous circuit specifications, which is an important phase of asynchronous design flows. The purpose of such an analysis framework is twofold: it helps designers make informed decisions at each iteration of refinement, and it paves the way for auto-

[1] 'Asynchronous' design is synonymous with 'self-timed' design.

[2] Other reasons include: skepticism "It will never work," ignorance "It is too difficult," and irrationality "Asynchronous VLSI is the Devil."

matically and efficiently exploring otherwise intractable design spaces of equivalent programs. The strengths of our framework lie in the flexibility and re-usability of analysis primitives and procedures for rapid analysis development, and interactivity which allows users to dynamically adjust queries based on information from earlier analyses.

1.1 Preliminary Background

Before we jump into the design flow of asynchronous circuits, we give a brief overview of how asynchronous circuits work, and why this design methodology is worth pursuing over traditional synchronous design.

1.1.1 What is Asynchronous VLSI?

A synchronous circuit is one whose activity is driven by a global clock. During each clock cycle, circuits evaluate their outputs as logical functions of their inputs. On each clock edge that demarcates each cycle, signals are latched and held for the duration of the next cycle, when they are re-evaluated. Thus a global clock orchestrates evaluation and latching in alternation, causing computed data to march along to a single beat in lock-step.

An asynchronous circuit lacks a global clock altogether. Its activity is driven by communication at its boundaries using local handshakes, which follow signaling protocols that indicate when local activity may safely proceed, and when it is complete. Instead of working in lock-step, computation and communication are entirely *event-driven*.

One naïve performance metric for a synchronous circuit is its global clock frequency. The clock frequency is limited by the slowest path(s) through logic be-

tween registers, known as the *critical path*. Violation of this constraint may result in the circuit operating incorrectly if signals are latched before their evaluation is finished. The challenge of optimizing synchronous circuits is in shortening the critical paths as much as possible, or meeting a target frequency, known as timing closure. Reaching timing closure can be an expensive phase of synchronous design verification because it is intertwined with the physical design phase which determines actual path delays (Section 1.1.2).

Asynchronous circuits are evaluated using their throughput, the average rate at which a unit of work is done, where each iteration may take different amounts of time. Critical paths in asynchronous circuits are more subtly defined: asynchronous performance is only determined by paths that are *actually* exercised at run-time, as opposed to paths that *may* be exercised in the synchronous counterparts. This is why asynchronous circuits are said to achieve *average-case* performance, rather than worst-case performance. For example, in a feed-forward, asynchronous pipeline without conditional paths, the critical path is simply the slowest component because all paths are exercised. In a different scenario, a slow component that is rarely used will have little negative impact on an asynchronous system's overall performance.

1.1.2 Impact of timing on design

To understand where design flows for synchronous and asynchronous circuits differ, we examine the role of timing in both design families. In synchronous design, timing plays a role from the beginning to the end: a specific clock frequency is targeted, and pipelining and register retiming is determined as a result of initial critical path estimation. *Static timing analysis* (STA) is performed throughout the design process to verify that the global timing constraints can be satisfied. A

design that is insufficiently pipelined may fail to meet the target frequency because paths between clocked registers are too long. An overly pipelined design will waste area and energy on registers, add unnecessary cycles of latency on certain paths, and may complicate register retiming. A timed specification is then synthesized to *register transfer logic* (RTL), which specifies the logical functions between clocked registers. RTL is synthesized into gate-level netlists, which in turn, beget transistor netlists. Each step of synthesis lowers abstraction and adds detail, providing better estimates (but no guarantee) of the actual path delays. The physical design phase (layout geometry and mask design) further increases timing accuracy with extracted electrical parameters and delays from analog simulation such as `spice`. Placement and routing of circuits should account for delays introduced by wire length and loading.

Other important provisions in timing validation include (but are not limited to): test pattern coverage to generate input-dependent timing signatures of subcircuits, and false path elimination to exclude impossible paths from consideration (which might otherwise exacerbate the worst-case delays). Timing constraints are further exacerbated by variability in the fabrication process, degradation, and variations in operating conditions (temperature, supply voltage, and noise), requiring designers to accommodate additional timing margins. If at any point during synthesis, timing closure is deemed unachievable, then the previous step must be revisited. When a timing constraint is not met during operation (through design error or external cause), some internal signal may be mis-evaluated, potentially causing a visibly wrong result or other silent malfunction. In spite of these challenges with synchronous design, there exist a long legacy of synchronous design tools (and immense labor and capital investments) in the industry to sustain and support the incumbent design methodology for future generations.

Asynchronous circuits liberate designers from having to continually mind the clock. The fact that asynchronous circuits can work correctly with arbitrary, finite gate delays[3] decouples functional correctness from performance optimization. One nice consequence of this separability is that a large class of asynchronous designs[4] can alter the physical pipelining (to improve performance) *without* affecting the logical pipelining (correctness). The role that timing plays in asynchronous designs is in performance optimization. Asynchronous design families that do utilize timing constraints, however, only do so *locally* in handshakes and communicating processes without imposing upon any global constraints.

Removing timing constraints from the correctness equation makes it possible to formally verify successive refinements of concurrent programs; each applied transformation is *mathematically proven* to be semantic-preserving. The ability to prove correctness of refinements pays off in an undeniable reduction in design time and effort, as demonstrated by small academic teams producing complex asynchronous chips working in first silicon [19, 20, 46, 47]. Verifiability and a short design time should not be undervalued, especially with the growing size and complexity of integrated circuits!

The ability to rewrite provably correct concurrent program specifications throughout the design flow is paramount to asynchronous circuit design practice. Asynchronous circuit synthesis can benefit from work in the field of compilers: program rewriting is one form of source-to-source translation, and circuit synthesis is the result of abstraction lowering. Our work described here aims to assist the program rewriting process, be it manual or automated. To recapitulate, the absence of timing from the formulation of correctness in asynchronous circuits has far-reaching implications on its design flow. In the next section, we describe our concept of an

[3]the quasi-delay insensitive (QDI) family
[4]slack-elastic designs[42]

asynchronous synthesis flow.

1.2 Asynchronous Design Flow

Both synchronous and asynchronous designs take high-level inputs and eventually produce low-level circuit netlists. Synchronous designs start with high-level behavioral descriptions in a language such as Verilog or VHDL. The initial description is eventually synthesized into RTL and then a circuit netlist, at which point it is handed off for physical design (always minding the timing constraints).

Most existing and proposed asynchronous design tools follow a flow that takes some variant of CSP as input and eventually produces a netlist of circuits. Figure 1.1 is our own rendition of a typical asynchronous synthesis flow diagram. Our asynchronous flow (not unique) begins with a high-level functional description in CHP, a variant of Hoare's CSP [25]. CSP and CHP feature semantics for explicit sequencing, message-passing, flow control, and concurrency. Appendix A provides a quick reference to CHP notation and semantics.

Early pioneering work in asynchronous VLSI showed that asynchronous circuits could be systematically synthesized from an abstract specification in CSP [44]. Syntax-directed translation (SDT) is a method of synthesis where the syntactic constructs map directly into asynchronous circuits that implement the primitive semantics [5]. With SDT, the quality of the resulting circuits depends on the characteristics of the source program; a program written sequentially will produce sequentially operating circuits. Section 2.3.1 discusses conventional techniques for synthesizing asynchronous circuits from high-level concurrent programs: dataflow-driven translation, template-based translation [6, 12, 68, 77]. Dataflow techniques succeed at decomposing larger blocks of code into a set of primitive nodes. Template-based translation is suitable for pattern-matching against sets

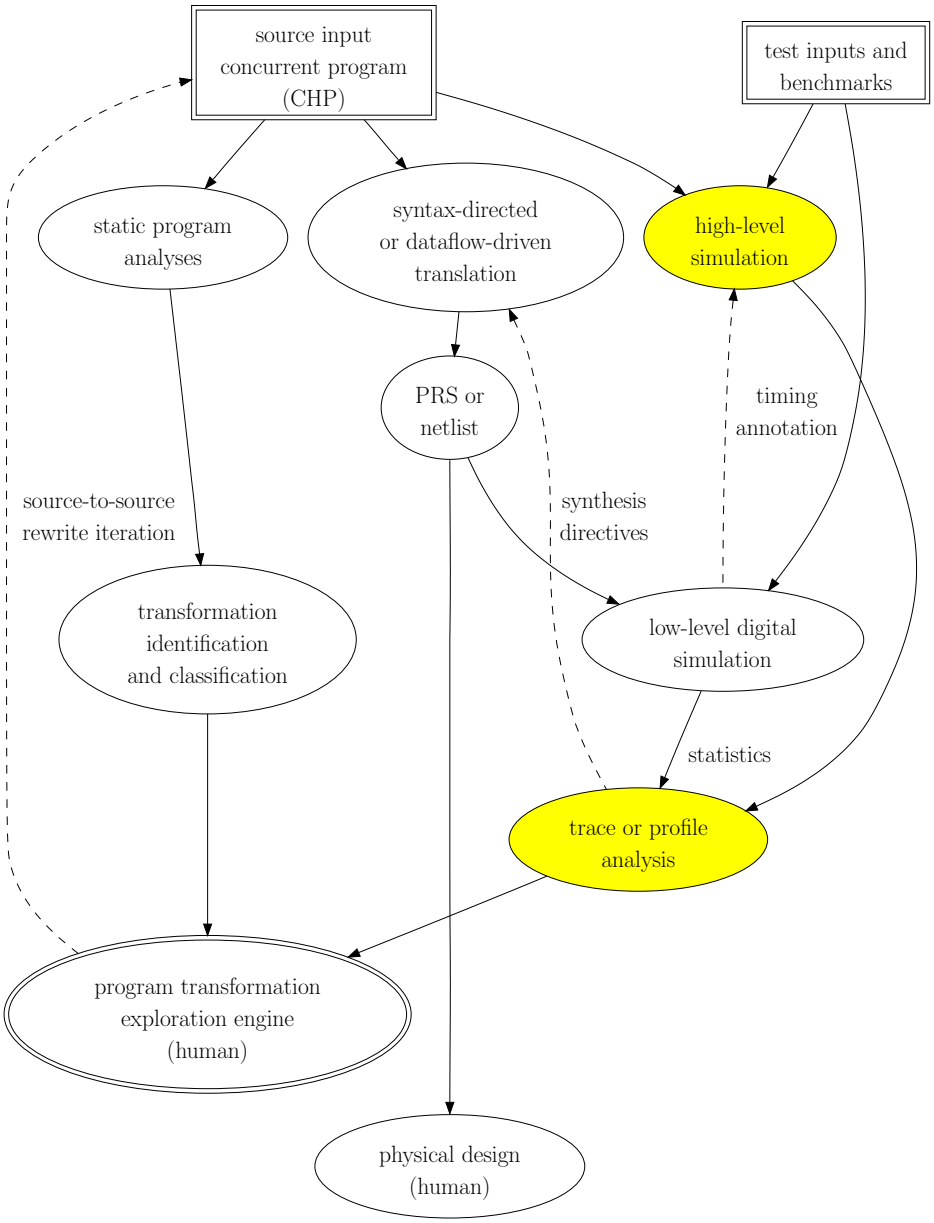


Figure 1.1: Asynchronous circuit synthesis flow

of known constructs and their corresponding circuits. While synthesizing circuits from programs using any of these techniques produces correct circuits, the resulting circuits are unlikely to be optimized because the input programs may lack explicit parallelism. Concurrency is attained by rewriting the source program explicitly into parallel, communicating processes. In theory, such rewriting can be done

automatically with the aid of a source-to-source compiler using static program analysis. In practice, rewriting is often done by hand.

Since the quality of circuits produced by a direct translation depends strongly on the input (of which there may exist numerous equivalent versions), it behooves a designer to optimize the input of the synthesis phase. Without proper analysis tools, a designer often relies on experience when evaluating the consequence of each high-level, structural transformation. Figure 1.1 emphasizes the use of *feedback* between various phases of synthesis[5], represented by dashed edges. For example, back-annotating timing from lower level simulations to higher level simulations improves the accuracy (while retaining high-level simulation efficiency) and aids in timing verification [33]. Providing feedback directives (say, derived from profiling) to the circuit translation phase may result in better choices in circuit optimizations. However, the impact of such optimizations cannot compete with the potential available from *restructuring* the concurrency and control flow of the circuit. Our primary goal is to aid rewriting high-level, concurrent program specifications of asynchronous circuits. Transformations applied to the source program should be justified by the expected benefits on a given set of inputs.

The other inputs to the design flow are test workloads or benchmarks. The test inputs should reflect typical conditions and data that the circuit is expected to encounter; for optimization purposes, they serve as a training set. Either high-level or low-level (post-synthesis) simulations can be used for comparison. For the work described in this dissertation, we use a high-level simulation of concurrent programs to assess each program's potential to produce optimized asynchronous circuits (Section 3.2). A high-level simulation allows one to evaluate the merits of the *structure* of a concurrent program without assuming details of how the

[5]By 'synthesis', we mean lowering the of level of abstraction while increasing the level of detail of specification.

resulting circuits will be synthesized. Program analysis and transformation at a high level is agnostic with respect to the specific asynchronous circuit family used in synthesis, and is thus, widely applicable to all asynchronous circuits design flows. The simulation and analysis framework we present is geared towards assisting rewriting and high-level optimization of concurrent programs.

1.3 Challenge and Contributions

> *There's more than one way to do it.*
>
> Perl motto and philosophy

> *"I used to write in Perl a lot. Nowadays Perl scares me. It looks like an explosion in an ASCII factory."*
>
> Diederik van der Boor,
> `dot.kde.org`, 2007-07-04
> (and *many* others before him)

The major challenge of rewriting or restructuring programs is that the space of functionally equivalent programs is unbounded; it is impossible to consider all equivalent versions of a program, simply because there are infinitely many "obviously poor" legal transformations. Compiler writers realized long ago that aggressive optimization is an iterative process. To know where one should spend effort on optimization and detailed analysis, one should concentrate on the most frequent paths and the most critical paths as found by (simulated) execution. Hotspots and critical paths indicate where transformations are likely to have the greatest impact, and can be used to prioritize program rewriting iterations.

Apart from the obviously good and bad transformations, the benefit of a transformation often depends on the local context in question, and moreover, the inputs and circumstances under which a piece of a program is executed. Chapter 4 gives

many examples of transformations whose benefit *depends on run-time conditions*, not deducible from any static analysis. Many candidate transformations exhibit a tradeoff between metrics such as performance, energy, and area. The designer (or source-to-source compiler) has the daunting task of choosing which transformations to apply. Simulation feedback and profiling will ultimately justify these decisions. Coupling static program analyses with run-time profile analyses increases the potential to effectively and aggressively restructure high-level programs and apply low-level synthesis optimizations [34]. The job of our analysis infrastructure is to make program evaluation more accessible, informative, and flexible to users.

Our contribution to the asynchronous circuit design community is a high-level simulation trace analysis framework, intended to help designers make informed structural optimizations of asynchronous systems, especially where optimization is non-obvious. Our overall infrastructure includes a concurrent program compiler for the CHP language, and an event-driven simulator, from which traces can be produced for detailed run-time analysis. The analysis framework provides an interface for viewing and mining trace information useful to the designer, and complements static program analysis for choosing program transformations. This dissertation describes in detail the analysis primitives and procedures available to the user, and demonstrates how useful analyses are constructed within the framework. We describe several cases of design choices where run-time analyses reveal strengths and weaknesses (not statically inferable) that can be exploited for program optimization.

1.4 Outline

There is much room for development and improvement of asynchronous circuit design tools. The design flow we propose touts program rewriting as having an

important role in design: gradually refining the concurrent program so that circuits produced by direct synthesis methods will be *structurally optimized* with respect to a given set of workloads. Program rewriting is orthogonal to all other phases of asynchronous circuit synthesis. The difficult problem of evaluating rewritten programs is aided by our new simulation trace analysis framework, which provides the means to extract detailed performance feedback from any execution. This dissertation describes the analysis framework and demonstrates its benefits.

In Chapter 2, we discuss related work in asynchronous VLSI that precedes our own, covering parallel program evaluation and existing asynchronous design tools. Chapter 3 describes our high-level simulation and trace analysis infrastructure, and justifies our approach. Chapter 4 describes how analyses can be constructed to assess the merits of other concurrent program transformations. In Chapter 5, we present some case studies that further demonstrate the utility of our analysis framework. Chapter 6 concludes this dissertation.

CHAPTER 2

BACKGROUND: RELATED WORK

The work for this dissertation spans several fields: parallel programming, performance analysis, asynchronous circuit synthesis. We start with a brief overview of parallel programs and their relation to asynchronous circuit design. The majority of our work builds upon ideas from performance evaluation of software parallel programs. Lastly, we summarize the current state of existing asynchronous circuit synthesis tools, and some of their attempts to incorporate performance profiling.

2.1 Concurrent Programming Languages

Parallel programming has roots in both the software community and in hardware design. Digital circuit design could be construed as one form of parallel programming: synchronous circuits change state at the beat of a global clock as determined by the logic between clocked latches, while asynchronous circuits compute and communicate in an event-driven manner using local handshakes on channels. Explicitly expressing concurrency is very befitting for hardware descriptions languages.

Hoare's Communicating Sequential Processes (CSP) is one such language for expressing concurrency and communication [25, 26]. In CSP, processes communicate data by passing messages over channels; send-receive action pairs (over the same channel) are synchronized point-to-point, i.e. when one of them is reached, it waits until its counterpart (possibly in another process) is reached before both sides proceed. (Send and receive actions are *atomic* and *blocking*.) Communicating Hardware Processes (CHP), a close variant of CSP, was used by Martin to compile parallel programs into delay-insensitive circuits [44, 45]. A quick summary of CHP can be found in Appendix A. We mention other synthesis methods in Section 2.3.

Other languages used for asynchronous circuit specification and synthesis share

essentially the same semantics as CSP, dressed in different syntaxes. Variants of CSP provide additional constructs that make concurrent hardware descriptions more convenient or more expressive. Tangram [73] and Balsa Synthesis Tools [12] are similar to each other; the latter is a publicly available variant of the former, which is proprietary. Balsa contains some flow control constructs that are not in CHP, (e.g. sequential if-else) but can be expressed (perhaps less conveniently) using CHP primitives. We have extended our own implementation of CHP with syntactic loop-expansions of repetitive constructs for convenience. TAST, from TIMA, features its own CSP variant (not publicly available) [60]. Haste is the CSP variant used by Handshake Solutions [55]. Occam adds the abstraction of dynamic process lifetime, with process instantiation and termination, which is suitable for abstraction in parallel software [69]. The abstraction of process lifetime may be useful for dynamically reconfigurable hardware, such as asynchronous FPGAs (AFPGA).

The purpose of these languages is to specify the high-level behavior of the asynchronous circuits to be synthesized, without getting involved in the implementation details such as channel encodings and handshake protocols. We use CHP as the high-level language in our design flow because it is simple, and has worked sufficiently well in the past. The choice of language is not pivotal to our analysis infrastructure; the key concepts in our simulation analysis framework are applicable to any CSP-like language.

2.2 Performance Evaluation of Parallel Programs

Analysis of parallel programs and hardware share a common purpose: to identify performance bottlenecks. Many techniques for performance evaluation of parallel programs inspire similar approaches to evaluating parallel hardware. The following attributes of parallel performance analysis systems are usually desired [21]:

1. **abstraction** — The ability to reason about events at a higher level given low-level details of execution helps users comprehend large volumes of information more easily.
2. **transparency** — The ability to measure a system without perturbing the measurement itself is valuable for accuracy.
3. **interactivity** — The ability to adapt and modify analyses based on observations enables users to iteratively and efficiently experiment with different solutions.
4. **portability** — Techniques should not be constrained to a particular model or implementation. Since there are different concurrent programming languages, analysis tools that support different languages variants would be more valuable than those limited to only one language.

2.2.1 Measurement and Tracing

Next, we explain how these traits have manifested in parallel software analysis, and how some of the same techniques carry over to hardware analysis.

Instrumentation and sampling. There are several ways of benchmarking parallel programs beyond just measuring execution time. By instrumenting a program with measurement code (in source or binary image), one can generate partial traces of detailed measurements for online or offline analysis. Since instrumentation often perturbs and prolongs the timing of program execution, it comes at the expense of measurement transparency. The convenience of instrumentation comes at the cost of accuracy, depending on the invasiveness of modification. The Paradyn performance analysis tool reduces measurement perturbation and the volume of traced data by *dynamically instrumenting* the executing program and non-

invasively sampling counters updated by the instrumented code [48, 50]. Pablo is an earlier portable and scalable analysis environment capable of dynamically adjusting the *level* of instrumentation using counter thresholds for feedback [58]. The pC++ performance analysis environment featured both runtime analysis and offline trace-based profiling [40]. More computationally expensive work was typically reserved for offline evaluation. A benefit of dynamic instrumentation is that queries can be constructed and deployed at *run time*, enabling dynamic experimentation and refinement of measurements. Periodically sampling measurements of a running program can be less intrusive than instrumentation, but may overlook details that are key to the understanding performance problems [24, 51, 52]. The 'state' of parallel hardware is not representable as call-stacks, but rather, a distributed set of program points. Thus, the stack-sampling approach taken by the `gprof` sequential program profiler does not fit the concurrent hardware model [22].

Simulation tracing. *Simulating* an executing parallel program, however, essentially decouples measurement from execution. Simulation affords the ability to trace every event in detail without perturbing the simulated execution (transparency), at the cost of trace storage. Alternate approaches trace only what is required to perform the desired analyses [29]. However, one does not always know *a priori* what information should be exacted before a program is executed; observations can inspire new avenues of investigation [21]. Trace storage in our design flow is justified by the potential need for fine-grain details of execution. In our analysis infrastructure, all analyses and refinements are performed offline on saved traces. A full trace is re-usable across many analyses on the same run; a new trace is required only when the input program or the workload changes. The storage cost and file access performance overhead of tracing can be a drawback when only the simplest queries are desired. For example, lightweight, on-the-fly counters in-

strumented directly into the simulation to mitigate the need to write and read a trace file. The flexibility of tracing and offline trace analysis is appealing when the analyses demanded are more diverse, detailed, and complex.

Simulation replay. Another benefit of storing a full trace is that it enables efficiently replay of the simulation, potentially revealing details that are not recorded in the trace file. Seeking to arbitrary times in the execution history can be accelerated with incremental checkpoints embedded in the trace file. Full tracing is especially useful for animated visualization of event activity for debugging and evaluation [21].

Version database. Over the course of concurrent program design and evolution, a designer is likely to amass a large history of data for every set of analyses run on each revision and input set. In addition to instrumentation and measurement, SCALEA features a database for storing results of experiments from measuring various versions of a parallel program [72]. Maintaining a database of analysis summaries would be very useful for a design space exploration engine to compare across versions of refinements of a parallel program. We mention the use of databases because they would complement any analysis and measurement framework for an iterative design process.

The ability to obtain measurements from a simulation (as opposed to invasive instrumentation or sampling) gives us the freedom of performing arbitrarily detailed analyses on execution traces, at the cost of trace storage. Our analysis infrastructure relies heavily on the simulation aspect of our design flow.

2.2.2 Program Simulation

We briefly mention a few simulation environments from which we draw principles for our own simulator. OCCARM is an Occam simulation model of the Amulet1

asynchronous ARM processor [69, 70]. The OCCARM simulation environment featured a monitoring process for collect profile information: occupancy, utilization, throughput, and other internal process state. Measurements are taken according to models of the concurrent processes and explicitly communicated to the monitoring processes. Rather than pass information to a monitor process, our simulator logs every atomic event and state change to a trace file as it executes.

EDPEPPS[1] is a design environment for portable parallel applications that featured a message-passing virtual machine simulator. The full-system simulator is organized in layers, spanning the hardware, operating system, message-passing layer, and running applications [11]. Since we target only circuit design, our discrete event simulator models only hardware. EDPEPPS includes numerous simulation, trace-analysis, and visualization modules, and is easily extensible and integrated with other tools and compilers. Our long term goal is to use our trace analysis framework to drive high-level program transformations in feedback-directed compiler optimizations. Eventually, support for mixed-mode simulation would allow one to map circuit-level events back up to a higher-level constructs.

We describe our simulator further in Section 3.2.

2.2.3 Trace Analysis

There are several existing tools for analyzing traces of parallel programs offline. One of the goals of our trace analysis framework is to provide an interface from which analysis libraries can be easily developed. Medea is one tool for processing trace files produced by monitors during the execution of parallel programs [8]. Medea provides a collection of statistical and numerical analysis modules, which are easy to integrate and coordinate with other tools. Their analysis of trace files

[1]Environment for the Design and Performance Evaluation of Portable Parallel Software

is aimed at constructing numerical models of performance as a function of input parameters, for performance prediction. IPS-2 also features a rich library of trace analyses [27, 49].

A proper interface is important for any trace analysis tool. Analysis tools often feature their own interface language to operate on traces or databases of experimental data [4, 63, 76]. Model-driven analysis systems use separate input languages to formulate analyses and queries at a *high level*, while a compiler automatically emits low-level instrumentation and analysis code [29]. We use Scheme as our interface language and provide an API consisting of primitive operations and predicates, thus leveraging all of the capabilities and functionality of the host language (Section 3.3). A layered approach decouples the details of the trace file format from the implementation of the analysis library, leading to better portability.

2.2.4 Temporal Analysis

A large class of trace analyses examine the properties of a program over time (temporal analysis). Properties can vary from simple (e.g. value of a variable) to complex (e.g. comparing activity factor between processes). The entire history of a trace can be classified according to such properties. A summary of time spent in each category or state is called a *time histogram*.

The Occam debugging environment described by Goldszmidt supported checking of temporal logic assertions on a program's execution history, suitable for debugging parallel programs [21]. The IPS-2 tools were capable of accumulating user-specified time histograms computed on traces [49]. The ability to construct arbitrary complex time histograms and queries is valuable for tailoring analyses to specific applications. Our trace analysis framework supports evaluation of arbitrary functions that sweep over traces for temporal analysis (Section 3.3).

Complex probe functions (run over a trace) can be formulated for debugging, performance and activity analysis. For example, one can approximate the dynamic activity factor by sweeping over all events with a fixed-size time window. One can verify the exclusiveness between two processes by 'monitoring' the state of the processes. Temporal analysis is the basis for detecting *phase changes* in programs. It is often desirable to partition the trace of a parallel program into distinct phases, and analyze or optimize each phase separately. Since the notion of phase can be very application-specific, it is important that users be able to apply custom functions to catch phase boundaries. Phase detection functions can be formulated by matching low-level activity patterns that translate to some high-level behavior [59].

2.2.5 Expert Systems Approaches

Expert systems is a field of artificial intelligence that utilizes some subject-specific knowledge of human experts in a knowledge base. The *inference engine* uses the *knowledge base* to present a series of questions to the user in an attempt to determine an answer in the subject domain. Some common examples of expert systems are found in technical troubleshooting and medical diagnosis. While we do not employ expert systems techniques in our framework, we adopt the concept of being able to extend a knowledge base of analyses and diagnostics for performance optimization.

Parallel programming and circuit design are domains where *experience* often helps with problem solving and diagnosis. Merlin is a tool for automating parallel program performance analysis that employs a knowledge base of rules that map performance symptoms to possible causes, and causes to possible solutions [34]. As a designer's experience grows, the knowledge base can be appended with new expertise in performance diagnosis, making Merlin useful to non-expert parallel

programmers. KAPPA-PI organizes its knowledge base into a hierarchy of possible performance bottlenecks from general rules down to specific rules [15, 16]. For example, a "frequently blocked sender" is a subclass of general "communication issues," and "barrier wait imbalance" can be a subclass of general structural problems. Typical analysis sessions start with the same core set of diagnostics, followed by different refined diagnostics with each iteration.

With our analysis infrastructure, one can construct an expert system for performance diagnosis by building a knowledge base of rules for diagnostics that trigger refined analyses and queries, which invoke routines from an analysis library.

2.2.6 User Interface Design

For analysis tools to be accessible to non-experts, it is essential to be able to present information in a structured (and often graphical) manner. Nearly every mentioned tool for performance analysis of parallel programs touts some graphical user interface and visualizations of analyses. Guidelines for effective interface design have been described in [54]. Although we have not developed any beautiful graphics along with our infrastructure, we do make information easily available to data visualization tools to leverage our work and others' work.

2.3 Asynchronous Synthesis Tools

Our review of other asynchronous circuit synthesis tools covers two aspects: methods for synthesizing circuits (lowering abstraction level from concurrent programs), and existing integrated design flows.

2.3.1 Circuit synthesis methods

Our work does not focus on circuit synthesis; however, understanding various approaches to synthesis gives some insight on how the results obtained from high-level trace analyses can direct better synthesis, and how high-level transformations may improve the results of synthesis. Syntax-directed translation (SDT) was the first approach used to synthesize asynchronous circuits from high-level concurrent program descriptions [6, 44, 45]. With SDT, syntactic constructs (such as sequence, concurrency, communication, and selection) are recursively mapped to circuits that implement these primitives. However, without fine-grain process decomposition, the resulting circuits would exhibit only as much concurrency as was explicitly written. *Projection* was introduced to partition variable assignments into send-receive pairs to facilitate process decomposition [41]. Process decomposition results in smaller and simpler processes capable of achieving greater throughput.

Once concurrent processes are factored into primitive processes, they can be handed off to circuit synthesizers. A. Lines described a method for synthesizing pipelined quasi-delay insensitive (QDI) circuits using known templates for common four-phase handshake protocols [38]. By changing relatively few subcircuits to implement different functions, a designer can easily write netlists, even by hand. The circuit templates can be chosen based on size, latency, throughput, energy, and scalability. This method of synthesis is typically reserved for the leaf cells of finely decomposed concurrent processes. Since there can be more than one way to synthesize circuits, this translation step can be guided by hints from both high-level and low-level performance profiling. For example, a non-critical path may favor smaller circuits for saving area without compromising performance.

Petri Nets (PN) are also commonly used to describe handshaking protocols and for synthesizing asynchronous circuits with tools such as Petrify [10, 35, 36, 37].

Since there are several handshaking protocols to choose from and multiple correct implementations for each protocol, run-time profiling of the high-level program can help the above synthesis methods decide which implementation is more suitable, depending on path criticality.

The ACK synthesis tool (no longer maintained) targeted synthesis of datapaths with (one or more) separate controllers from high-level descriptions, initially in Verilog or VHDL [31]. The controls for the datapath are synthesized as asynchronous state machines, using control graph partitioning to simplify synthesis. The computation portion of the datapath leverages standard synchronous (VHDL) back-end synthesis. The TiDE tools translate Haste program descriptions into bundled-data style circuits with a 4-phase control handshake, which requires separate timing validation of delay elements [64]. The leaf circuits are mapped to a standard cell library, chosen by the user, using standard EDA tools. The "different where needed, standard where possible" mantra, where existing synchronous tool flows are used to a great extent, is prevalent among several asynchronous synthesis tool chains — circuit synthesis may not be optimal, but they provide a short path[2] to a working design flow.

Message-passing concurrent languages fit extremely well with the conventional dataflow framework in optimizing compilers. Data-driven (or dataflow-driven) synthesis is another approach that produces circuits in a manner purely dependent on data dependencies [68, 77]. Data-driven synthesis produces finely decomposed and deeply pipelined processes, sometimes *overly pipelined* on latency-critical paths. One proposed solution was to apply sequential and parallel clustering algorithms to un-pipeline selected processes [78]. Clustering is also applicable to FPGA-style synthesis, where a computation is mapped onto a logic fabric with fixed resources [56]. Where static analysis runs into limitations, clustering heuristics

[2]low *development* cost, time, and effort

would benefit from guidance based on profile analyses of simulated executions.

2.3.2 Existing tool flows

The methods described above have been harnessed in tools developed in academia and industry. We summarize the simulation and analysis capabilities of some of those tools.

The Balsa Synthesis Tools, based on Tangram, follow template-based, syntax-directed synthesis, and provide a library of primitive components [12, 73]. Balsa includes a simulator that produces a trace of channel and process activity for profiling [13]. Balsa provides a variety of visualizations to aid in debugging and analysis, such as deadlock causes. The Balsa designers realized the importance of being able to iteratively refine a source description of a concurrent program to optimize the circuits produced by syntax-directed translation [32]. However, the provided analyses are very rudimentary and not easily extensible[3], and there is little headway towards assisting program rewriting using the existing analyses.

Profiling simulation of asynchronous circuits already exists in synthesis tools. The TIMA Asynchronous Synthesis Tool (TAST) includes an activity profiler capable of collecting frequency statistics about execution paths and channel/variable data [60]. The activity profile acquired using TAST or similar tools is intended to guide optimizations:

- Area and energy can be reduced by eliminating unused circuits, or downsizing infrequently used circuits.
- Performance may be improved by scheduling more aggressively on frequent critical paths.

[3] source code editing and compilation required

- Electromagnetic emissions can be reduced by scheduling activity more evenly over time.

One particularly novel application of the TAST profiler used frequencies of a multi-way selection to implement an unbalanced multi-level selection favoring the most frequent case (which they called "choice structure optimization"). Similar optimizations were done by hand in the design of the MiniMIPS' datapath, Lutonium datapath, and sensor network asynchronous processor (SNAP) datapath [30, 43, 47]. Our approach to profiling is to provide an *extensible* framework from which arbitrary analyses can be constructed and run on saved execution traces.

The Timeless Design Environment (TiDE) tools from Handshake Solutions synthesize self-timed circuits by mapping the high-level program (in Haste) into a Verilog netlist in a bundled-data style (separate control and data) and uses standard commercial tools for back-end synthesis [64, 66, 71]. TiDE's simulator interfaces to a graphical interface for basic performance and coverage analysis, but the interface lacks the ability to develop and extend analyses. Their synthesis flow supports low-level synthesis directives, such as handshake protocol selection, at the source-level, however such annotations are not yet automated. The design flow does not provide a mechanism for feeding the results of performance profile analysis to other parts of the tool chain. TiDE's simulation environment supports behavioral-level (Verilog) and handshake-level execution modeled in C. In contrast, our high-level simulator, `chpsim`, models handshakes as point-to-point synchronization in event-driven, data-driven execution (Section 3.2).

Nielsen, et al. presented a synthesis flow that more closely resembles that found in synchronous design flows [53]. Their front-end converts a behavioral description (in Verilog, VHDL, or System C) into a control dataflow graph (CDFG) inter-

mediate representation, which is then translated into a CSP-like language for the asynchronous back-end for syntax-directed synthesis in Balsa [12]. Since the back-end is a straightforward mapping to circuits, Iterative refinement is done at the CDFG level. They evaluated the effectiveness of synchronous synthesis techniques on a fundamentally asynchronous representation. Their synthesis featured automatic resource sharing and constraint-based design space exploration of low-level circuits, algorithms for scheduling, allocation, and binding, and suitable mapping from CDFG to Balsa handshake templates. The phases that would best utilize run-time profile information are the CDFG transformations, and the design exploration of low-level synthesis.

2.3.3 The common ground and missing link

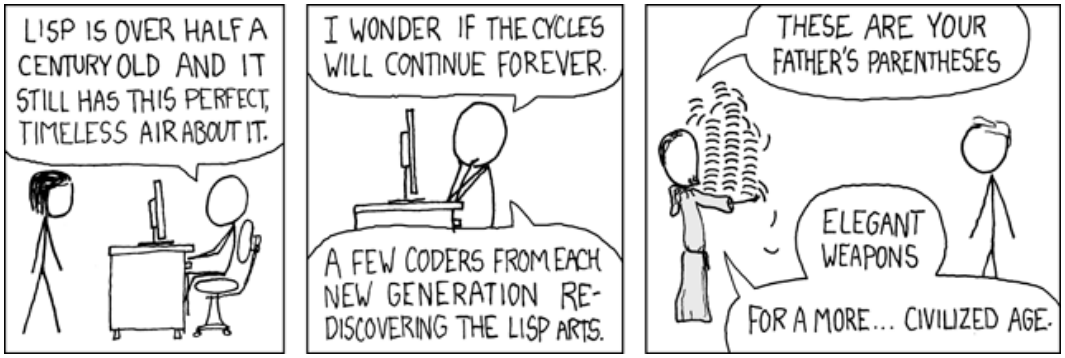
One common theme surrounding many asynchronous design flows is that *effective circuit optimization is an iterative process*. Many tools feature simulation profiling and analysis to help designers identify bottlenecks early in the design phase. However, those analysis tools lack a flexible interface for constructing and programming arbitrary analyses for reuse. Our contribution provides extensible profile analyses from the simulation of high-level concurrent programs and a functional programming interface to work with trace data. The resulting profile information can be used to drive subsequent iterations of high-level program transformations, initially through human interaction or eventually through automated compilation.

Much effort in asynchronous synthesis flows has been devoted to mapping of concurrent logic to handshake components and circuits. However, one aspect that is missing from asynchronous design flows is high-level program restructuring and rewriting. Aside from process decomposition, there has been little work the source-to-source transformation, which has a great potential to yield structurally opti-

mized circuits. The dissertation continues with the description of our simulation and profile analysis infrastructure (Chapter 3), and a survey of local program transformations that can exploit run-time profile information.

CHAPTER 3

PERFORMANCE EVALUATION INFRASTRUCTURE



Randall Munroe, `http://www.xkcd.com/297/`

This chapter describes the simulation and analysis infrastructure for evaluating asynchronous circuits at a high level of abstraction. First, we briefly describe the concurrent program compiler. The second part describes the execution model of the high-level asynchronous system simulator. The third part describes features of the trace structure along with primitive trace query operations. Chapter 4 then shows how analyses constructed within our framework can be used to aid program transformations.

3.1 Language and Compiler

The Tao gave birth to machine language.
Machine language gave birth to the assembler.
The assembler gave birth to the compiler.
Now there are ten thousand languages.
Each language has its purpose, however humble.
Each language expresses the Yin and Yang of software.
Each language has its place within the Tao.
But do not program in COBOL if you can avoid it.

The Tao of Programming

The input language used in our asynchronous design flow is named Hierarchical Asynchronous Circuits (HAC). The language supports common semantics found in HDLs: parameterized type definitions, type-safety, instances, arrays, and port connections. The key sub-languages are CHP for high-level concurrency semantics, and PRS (production rule set) for logic specifications. Sources can be (but need not be) compiled into object files that are used (and re-used) by other back-ends of the tool chain. For performance and memory efficiency, type and instance information is shared to the fullest extent.

The object file saves other tools (such as simulators) the effort of storing re-usable, stateless information. All of the type and hierarchy information (definitions, instances, variables) stored in the object file is accessible to the user through the same programming interface used for trace analysis (Section 3.3). Maintaining precise hierarchy is essential to the ability to associate expanded constructs with their origins in the syntax tree. The ability to trace back variables and events in the simulator back to the source is vital to constructing informative analyses.

3.2 CHP Simulator

The simulator, `chpsim`, is launched with an object file that encodes the whole program. Upon initialization, space is allocated to capture the entire state of the simulation: the whole program event graph, the values of all variables, and event queues. As the simulator executes events step-wise, the state is updated incrementally, and may be logged to a trace file for offline analysis. Our description of the `chpsim` simulator consists of three parts: the event model, execution algorithm, and tracing.

3.2.1 CHP Event Graphs

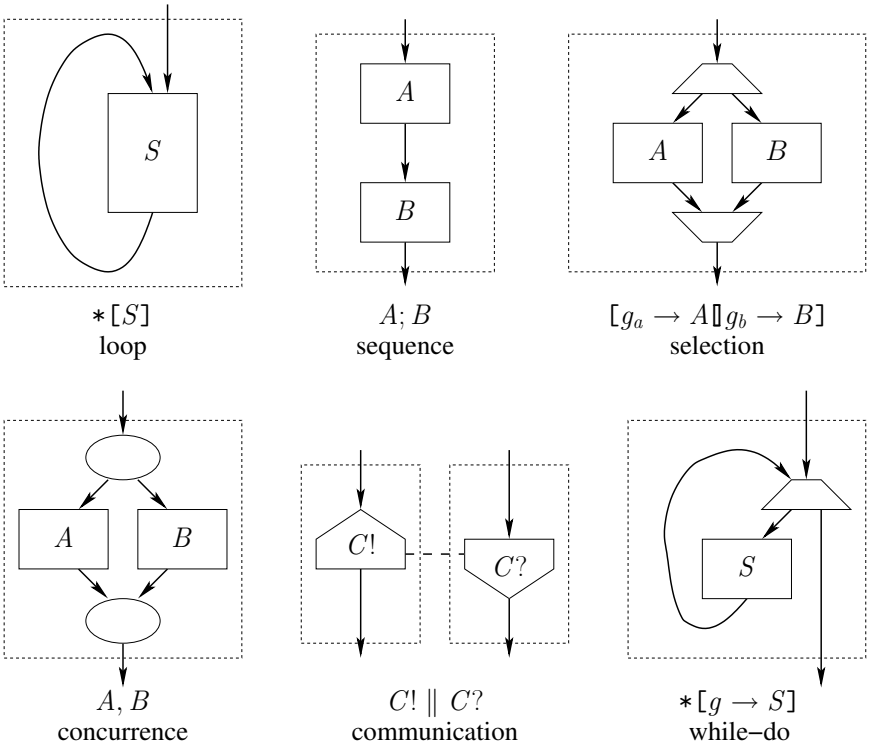


Figure 3.1: Syntax-directed translation of CHP to event graphs

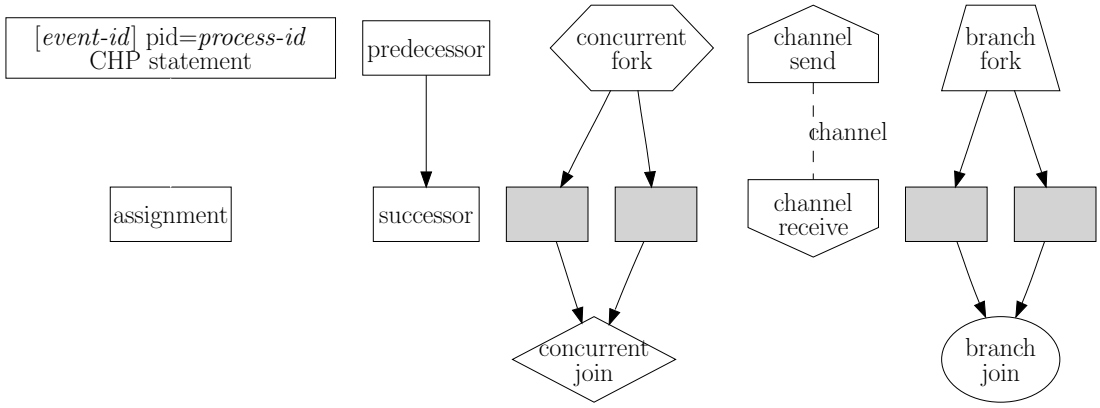


Figure 3.2: CHP event graph legend

The CHP source is expanded into a whole-program event graph (more specifically, a concurrent control flow graph) with a straightforward syntax-directed translation, analogous to circuit synthesis translations [6, 45]. Figure 3.1 sum-

marizes the translation mappings. A node in the fully expanded event graph represents an atomic action, and an edge represents a direct sequence relationship (edge (S, T) means S's completion *may* initiate T).

Figure 3.2 shows the legend for node shapes for all CHP event graphs in this document. For loops, the last event of the body points back to the first event of the body. Atomic events include assignment statements ($x := y$), condition waits ([G];), sends, receives, library function calls[1], and 'skip' actions. Sequences are just successor chains of events. Sequences that consist of only atomic events are analogous to basic blocks in conventional control flow graphs (CFG). Concurrent events initiate all successors in parallel and wait for all branches to finish before continuing. Deterministic and non-deterministic selections follow only one branch depending on the state of the guards. When no guards are true, the selection blocks waiting until a guard becomes true. Communication occurs when a send and receive event (in different processes) accessing the same channel are reached, at which point data is passed and both events complete. Sends and receives are *blocking* and serve as point-to-point synchronizations. The while-do construct is a two-way deterministic selection, with one branch entering the body (which returns to the selection), and an exit branch. Recursive expansion of these CHP constructs results in an event subgraph per process[2]. Each event node in a process subgraph contains a back-reference to the CHP syntax tree node that produced it, which directly maps any global event to its source (Section 3.2.3).

Each process contains one entry edge to its event subgraph, pointing to the starting program point for each process[3]. Unlike sequential programs which have

[1] `chpsim` features dynamic loading of plug-in libraries of user-defined functions in C++, made possible by GNU Libtool's libltdl.

[2] Actually, only one subgraph is stored per *unique type* because there are no interprocedural edges, while the simulation state maintains per-process graph markings. Processes only directly interact with each other through channels and shared variables.

[3] A process with explicitly concurrent sub-processes will immediately initiate multiple

only one active program point at a time, concurrent programs have at least one per process. Within a process, explicit concurrency begets multiple active program points (analogous to *threads* in parallel software). A single process can be visualized as a marked event-graph with one or more markers, and a concurrent whole-program can be seen as a collection of such processes. Event-graph markers may only move forward along edges, where the node type and current state determine which outgoing successor edges are followed. Simulation terminates when no events are able to execute and move forward.

The state of a CHP simulation includes a list of all *active events* (or marked) in the whole-program marked graph (typically few per process), and the value of all state variables, including channels. In the next section, we define what it means for an event to be active. The entire state can be checkpointed and restored to resume simulation later. A checkpoint file is associated with the same object file used to run the simulation so the checkpoint need not replicate hierarchy information. Checkpointing is useful for a simulation-based analysis infrastructure because it allows short simulations to be 'paused' and preliminarily analyzed before resuming selected experiments longer to collect more statistics.

3.2.2 Execution Algorithm

The simulator executes events in a purely event-driven manner, which naturally follows the way asynchronous circuits work. The 'lifetime' of a CHP event is divided into several states (Figure 3.3):

- **inactive**: The event is not queued for evaluation or execution. An event leaves the inactive state when all of its required predecessors have executed. An event becomes inactive immediately after it has executed. The only events that wait for more than one predecessor are the join events at the tail

events in parallel.

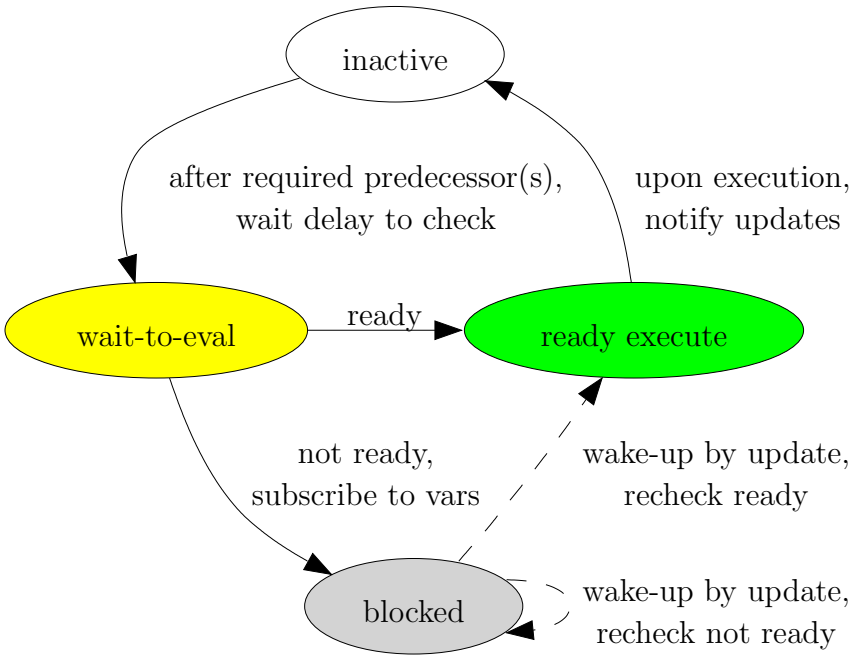


Figure 3.3: `chpsim` event life cycle

of every concurrent section; all others wait for only one. Most of the time, a majority of events will be inactive.

- **wait-to-eval**: The event is waiting for a delay to be checked for readiness (first time). This corresponds to a 'prefix' delay that is applied to every event. The time cost for each event is paid up-front during this phase. At the end of this period, if the event is evaluated ready, it is queued for execution.
- **blocked**: The event has been checked for readiness, but found not ready, so it must block and 'subscribe' to state variables whose value may affect its status. Condition-wait actions simply wait for the guard expression to become true. Communication actions may block on channels. Events that wake-up to value changes will either remain blocked or unblock for immediate execution.
- **ready-execute**: Event is ready to execute immediately. Events in this queue have already paid their delay, and are executed (one-by-one) with the same timestamp. Upon execution, an event becomes inactive.

Every occurrence of an event incurs its delay exactly once: before it is evaluated for execution for the first time (as it transitions to 'wait-to-eval'). Re-evaluations in the blocked state and the actual execution itself incur no additional delay. *Active* events are those in the 'wait-to-eval', 'blocked', or 'ready-execute' states.

Blocking and dependency subscribing. Events that are checked and found not ready to execute are said to *block* waiting. For example, send and receive events may be blocked by the channel(s) referenced in the communication action, and selections statements may be blocked by the variables in the guard expressions. A newly blocked event 'subscribes' to a set of dependent variables which can possibly unblock the event when their values change[4]. The subscriptions represent dynamic dependencies: the set of events subscribed to each variable changes at as events block and unblock. Each event that executes 'notifies' its modified variables (including channels) for re-evaluation. Events that are ready to execute are unblocked and placed in the ready-execute queue, while others just remain blocked and subscribed. Since inactive events are not subscribed to variables, they are never notified by variable updates.

To recap, events are evaluated for execution after they have waited a prefix delay. If the initial evaluation blocks execution, then the event subscribes to its dependent variables, otherwise it executes immediately. Subsequent re-evaluations (from wake-up on updated variables) that continue to block retain the same state, and unblocking events unsubscribe dependencies and execute immediately.

Channel sends and receives. Blocking sends and receives require that the state of a channel track which event arrived first and blocked. A sender that accesses an inactive channel will block until the corresponding receive is reached, and vice versa. After a send-receive pair of events execute, the channel returns to the inactive state. Figure 3.3 shows the state transition diagram for channels. Solid edges represent states changes that are caused by events local to the same process, and dashed edges represent state changes caused by events in other processes. The 'sent' and 'received' states are only momentary because send-receive pairs are

[4]The set of dependencies is computed statically and may be conservative with respect to run-time index values.

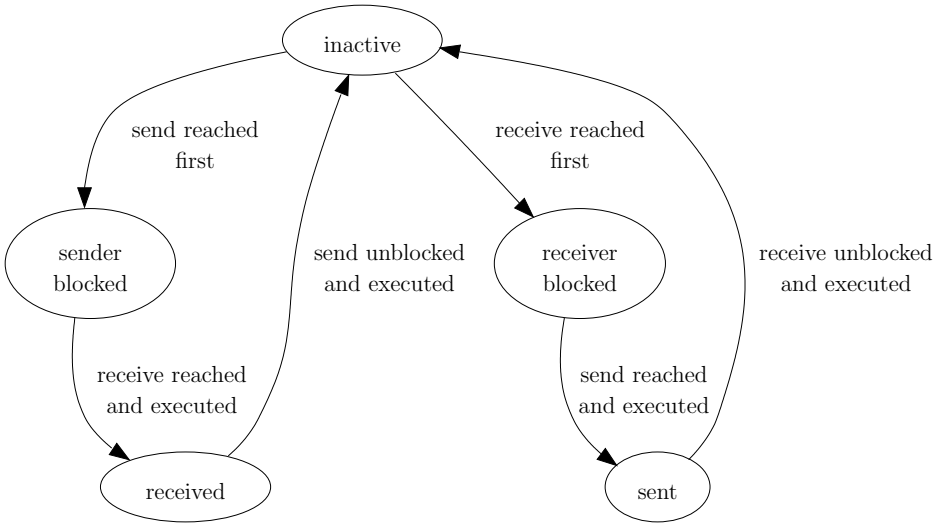


Figure 3.4: `chpsim` channel status changes

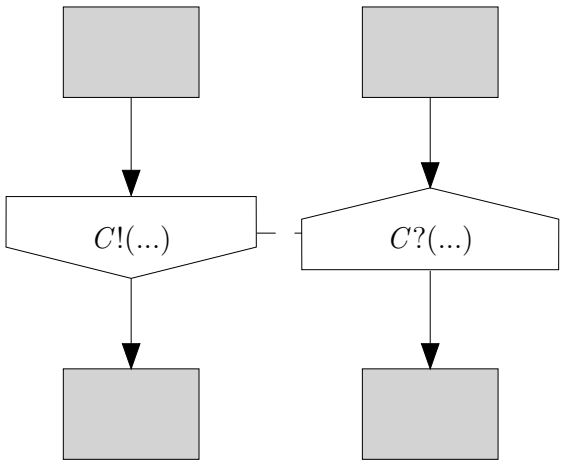


Figure 3.5: Communication over channels is simulated as a point-to-point synchronization between two processes; neither process can complete its communication action until its counterpart has also been reached.

guaranteed to execute at the same time (atomically); no other events can interrupt or separate them.

Our CHP implementation also supports some non-blocking semantics. A *channel probe* (denoted $\overline{Channel}$) is a boolean-valued expression that evaluates true if the referenced channel is in the 'sender-blocked' state, indicating that the channel already contains a value from the sender. Probing a channel allows choice of action

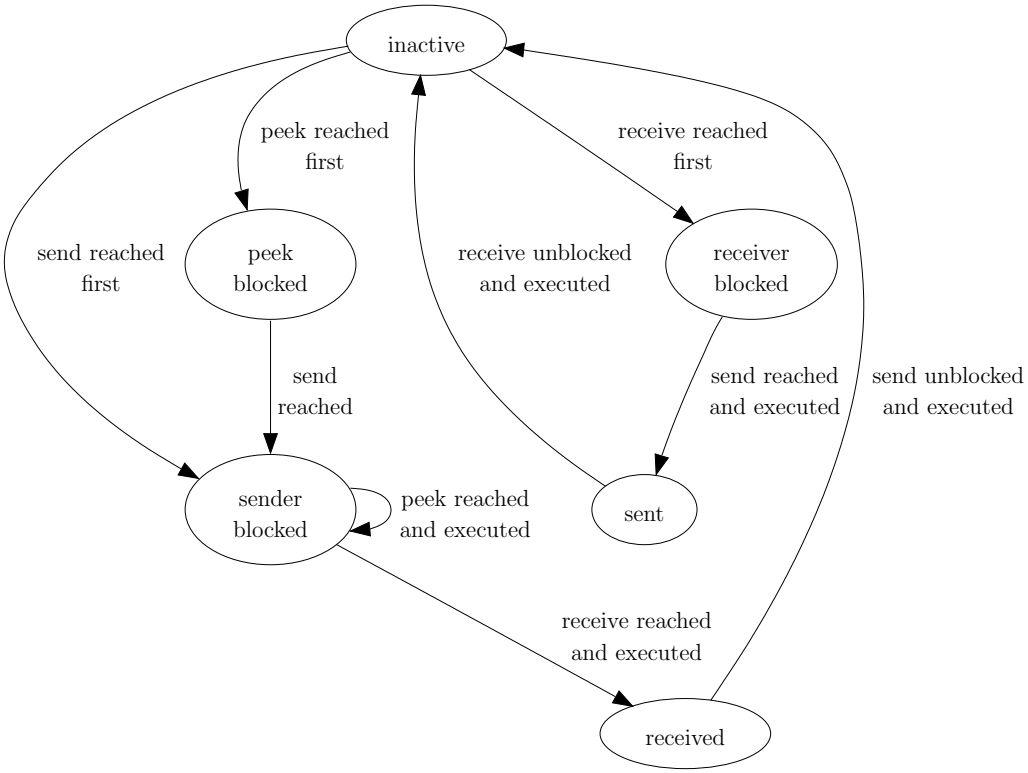


Figure 3.6: `chpsim` channel status changes, with peek

to depend on the current state of the channel, and can be used to express non-determinism. A channel 'peek' (denoted $X¿(x)$) reads the data in a sender-blocked channel into a variable without completing the receive transaction. Peeking allows action to proceed based on incoming values before completing a channel transaction. Peeking will block on an inactive channel as if it were a receive, but will not allow the sender to unblock when it is reached. The channel state transition graph with non-blocking actions is summarized in Figure 3.6.

Point-to-point synchronization on channels places some constraints on events that access channels: a single channel cannot support more than one outstanding send event and more than one outstanding receive (or peek) event. Violation of these exclusion constraints will result in a run-time error. Illegal state transition edges are not shown in Figures 3.4 and 3.6. However, it is legal for a channel to be

shared among multiple senders and receivers (even spanning different processes), as long as exclusive access to the shared channel is maintained.

Timing model. During event graph construction, each event is assigned its own constant delay, which is value-independent[5]. If a delay is not specified in the source, then it is given a default value based on the event type. Performance estimations from high level concurrent program simulation can always be improved by annotating delays from post-synthesis simulations. For the purposes of this dissertation, assuming a default set of delays in our simulations does not detract from the important concepts and contributions.

3.2.3 Execution Tracing

Tracing lies at the core of our analysis infrastructure. `chpsim` supports tracing for every event and state change in simulation. A user can also select which time intervals to trace if she knows *a priori* which intervals are interesting.

Our trace file contains two major components: an event trail, and a variable state trail. The event trail records each event as it is executed, noting its timestamp and global event identifier. Global event identifiers enumerate all events in the whole program event graph. Every global event identifier can be traced back to its parent process instance and its local event identifier within the process type[6]. The variable state trail records value changes with the global index of the event (position in the event trail) that caused the change.

Extensions. The event and variable trails represent a small set of information needed to perform a wide variety of analyses, but the trace format and interfaces can be extended with more information. One such addition that `chpsim` tracks

[5]Value-dependent delays and back-annotated delays are future work.

[6]This is possible because all processes of the same type share the same subgraph structure, whose events are enumerated locally.

is the *last completing predecessor event* that allowed each event to execute (or unblocks event). The last completed event facilitates efficient reconstruction of a precise critical path (Section 3.6).

Portability. The analyses we develop are bound to a set of event types which arise from CHP. However, many CSP-like languages used in asynchronous circuit design also map to the same set of primitive events. To leverage the same set of trace analysis tools from our framework from a different compiler and simulation environment, one has two options: either produce a trace file of compatible format (contents and file structure), or provide a different trace format along with an interface to access event trail and value trail information. The latter option takes full advantage of our Scheme environment described in Section 3.3 and is less invasive to development. Language-dependent features and analyses will require a different set of interface functions.

3.3 Analysis Environment

In designing an analysis environment, we desired the following (somewhat overlapping) characteristics:

- **interactivity**: Our infrastructure should provide an environment that gives the user full control over execution and the ability to inspect and manipulate data arbitrarily. This is typically achieved with a command-line interpreter. Analysis routines and command sequences can always be scripted for reuse and non-interactive use.
- **extensibility**: Users should not be restricted to the set of analyses developed by a single author. We emphasize the *ease* with which new analysis routines can be prototyped and developed with our infrastructure.

- **versatility**: An infrastructure that can adapt to traces produced by other tools is more valuable than one that is restricted to internal formats. The interpreter's programming environment should make it very easy to alter the view of foreign trace files to make them accessible to analyses in our native environment.
- **accessibility**: The ability to access information from the compiler, simulator, and trace files without developing a custom library saves the user from unnecessary development effort. The user should be free to export data from the analysis environment to other tools.
- **modularity**: The trace analysis library should not be closely coupled to the compiler and simulator; their development should remain as independent as possible. Modularity helps to maximize reuse of code by hiding implementation details across common interfaces.

With these traits in mind, we elected to use Scheme[7] as the host language for trace analysis development, along with GNU Guile's embeddable Scheme interpreter [23]. Scheme is a dialect of Lisp, both known for their powerful functional programming support [1, 74]. Scheme's philosophy is to provide a minimal set of language primitives from which rich libraries are developed. One notable difference from traditional Lisp is that variables are statically (or lexically) scoped, making function behavior more discernible at compile time.

With the need to query trace files for simulation information, one could have imagined providing a database interface to trace files. While databases do excel at pattern-matching and data-mining of records, their support for construction and application of higher-order procedures, which is vital to rapid development and reuse of analyses, is much more limited than that of functional programming

[7]Scheme was developed in the 1970s by Guy L. Steele and Gerald Jay Sussman, and is still widely used at this time of writing.

languages. Development of flexible analysis routines is aided by procedural abstraction. If a database query interface is desired, one can be constructed from functional languages [1].

Guile is a library for *language extension* [23]. It provides a bridge between C and the Scheme environment, in which dynamically loaded plug-in libraries (also known as modules) can interact. Extension languages are ideal for simplifying development by reducing cooperation effort between developers. The Guile interpreter provides an interactive interface to trace analysis primitives and routines. A plug-in architecture is convenient in that the baseline program can be extended without recompilation. The convenience of rapid analysis prototyping in an interpreted environment comes at a cost: the execution of analysis routines pay a run-time cost of interpretation. Should the need for performance arise, Scheme procedures can be re-written in C and compiled and run natively.

Understanding the examples and program listings in this document requires some basic knowledge of Scheme or other Lisp dialect. The interested reader is referred to the seminal Scheme text for an excellent exposition [1]. In addition to the primitives and procedure libraries included with Guile, we supplement the core library with generic algorithms and procedures in Appendix C.

3.3.1 Static object file queries

A large class of queries and operations only require static (stateless) information from the program source, such as type and hierarchy information. Static information gives additional meaning to the results of trace analyses by associating run-time events and observations with hierarchical structure of the concurrent program and its source. In this section, we describe of some query functions that extract information from only the object file, prior to any simulation. A more

complete list of basic object file queries can be found in Appendix D.

Some common query functions can be demonstrated with a few examples[8]. Consider HAC Program 3.1:

Program 3.1: Source connected to sink

```
// source−sink.hac
defproc source(chan!(bool) X) {
  chp { ∗[X!(true)] }   // send value in infinite loop
}
defproc sink(chan?(bool) X) {
  chp { ∗[X?] }         // consume value in infinite loop
}
chan(bool) Y;           // declare boolean channel Y
source A(Y);            // connect Y to source
sink Z(Y);              // connect Y to sink
```

Source defines a process that repeatedly sends a **true** value, and *sink* defines a process that consumes boolean values. In the Scheme environment, we can query some basic information about instances. An interactive session may look like the following:

```
$ hacguile source-sink.haco
hacguile> (define yref (hac:parse-reference "Y"))
hacguile> yref
(channel . 1)
hacguile> (hac:parse-reference "A.X")
(channel . 1)
```

An *instance reference* object is represented as a type-index pair, that refers to a globally unique instance[9]. Since Y is connected to A's port X, Y and $A.X$ refer to the same unique channel. One can query all aliases of any instance:

[8]These examples are run with a small test program, `hacguile`, which simply loads a compiled object file and provides an interface to internal data structures and the intermediate representation in an embedded Scheme interpreter.

[9]We also provide a *raw reference* Scheme object that captures the hierarchical structure of the referenced instance. Raw references can be manipulated by additional interface functions.

```
hacguile> (hac:lookup-reference-aliases yref)
("A.X" "Y" "Z.X")
```

Since the representation manipulated by the simulation trace analyses work with type-index pairs, this is useful for translating internal representations of references into human-comprehensible source-level references. One can retrieve type information about every instance:

```
hacguile> (hac:typeof-reference yref)
"chan(bool)"
hacguile> (define aref (hac:parse-reference "A"))
hacguile> (hac:typeof-reference aref)
"source<>"
hacguile> (hac:typeof-reference (hac:parse-reference "Z"))
"sink<>"
```

Basically, most static information about the structure of a concurrent program (in HAC) can be queried through these primitive functions. All object-file Scheme routines are also available in the post-simulation trace analysis environment.

3.3.2 Simulator structure and event queries

Even before a simulation is performed, one can collect information about the whole program CHP event graph, as constructed during initialization (Section 3.2). The environment for all queries and analyses related to `chpsim` is launched by running `hacchpsimguile`. We can examine every event in the whole program graph constructed by `chpsim` [10]. More primitive procedures related to the state of `chpsim` (pre-simulation) can be found in Appendix E. The following examples follow Figure 3.7, which illustrates the whole program event graph generated by Program 3.1.

```
$ hacchpsimguile source-sink.haco
hacchpsimguile> (define ep0 (hac:chpsim-get-event 0))
```

[10] We use a combination of helper programs, including Graphviz's `dot`, to automatically produce graphical output of event graphs.

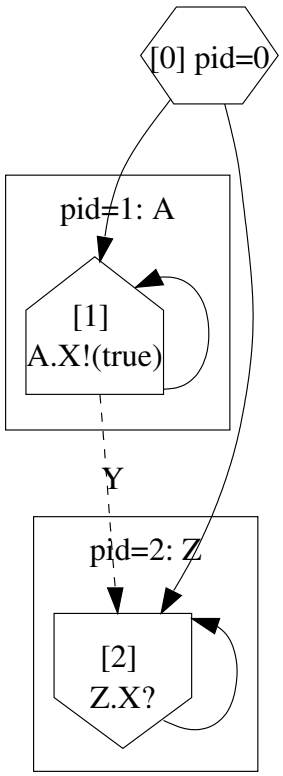


Figure 3.7: CHP whole program event graph for Program 3.1

```
hacchpsimguile> (define ep1 (hac:chpsim-get-event 1))
hacchpsimguile> ep0
(0 . #<raw-chpsim-event-node>)
hacchpsimguile> ep1
(1 . #<raw-chpsim-event-node>)
hacchpsimguile> (hac:chpsim-event-wait?  (cdr ep1))
#f
; is not a wait event
hacchpsimguile> (hac:chpsim-event-assign?  (cdr ep1))
#f
hacchpsimguile> (hac:chpsim-event-send?  (cdr ep1))
#t
; is a send event
```

We can query some properties about individual events with some primitive procedures.

```
hacchpsimguile> (define e1 (cdr ep1))
hacchpsimguile> (define e2 (cdr (hac:chpsim-get-event 2)))
hacchpsimguile> (hac:chpsim-event-num-predecessors e1)
```

```
1
hacchpsimguile> (hac:chpsim-event-successors e1)
(1)
; event's only successor is itself, event 1
hacchpsimguile> (hac:chpsim-event-successors e2)
(2)
hacchpsimguile> (hac:chpsim-event-process-id e1)
1
hacchpsimguile> (hac:chpsim-event-source e1)
"send:  *[A.X!(true)]"
; event 1 sends on channel A.X.
```

`hac:chpsim-event-source` is the key procedure that traces each event back to its precise origin in the HAC source. Many analysis routines will use this function to show the position in the concurrent program description in CHP, as the user had written it.

3.4 Static analysis procedures and variables

A small set of primitive operations provides a sufficient foundation for a variety of static program analyses. Static analyses are useful for discerning the characteristics of a program that are independent of inputs, including structural and flow information about the whole program. In this section, we give a quick tour of some of the core static analysis procedures built from the primitives. Listings for these procedures can be found in Appendix E.5.

The set of all events in the whole program graph is represented as a stream variable, *all-static-events-stream*. Many queries focus on a subset of events, based on event type or some other property. We define a higher-order procedure for filtering events using arbitrary predicate functions, `chpsim-filter-static-events` (Program E.1).

Specialized event filters can be defined by binding predicate functions, such as those listed in Section E.2. For example, a simple use of this filter is a search for

all selection events.

```
(define (chpsim-filter-static-events-select estrm)
  (chpsim-filter-static-events
    hac:chpsim-event-select? estrm))
(chpsim-filter-static-events-select
  all-static-events-stream)
; produces list of all selection events
```

A more complex event filter operation can be performed by passing composed predicates. The following example finds selection events with exactly two successor branches:

```
(chpsim-filter-static-events
  (lambda (e) (and (hac:chpsim-event-select? e)
    (= (hac:chpsim-event-successors e) 2)))
  all-static-events-stream)
```

Throughout this dissertation, we frequently focus on events that involve channels on the critical path. To identify all events that can affect a certain channel, we use the following procedure (Program 3.2):

Program 3.2: `chpsim-find-events-involving-channel-id`: Procedure to find all static events that can affect the state of a channel

```
(define (chpsim-find-events-involving-channel-id
    cid events-stream)
  (chpsim-filter-static-events
    (lambda (e)
      (any (lambda (i) (= i cid))
        (dependence-set-channels
          (hac:chpsim-event-may-block-deps-internal
            e))))
  events-stream))
```

`chpsim-find-events-involving-channel-id` filters out a set of events using a predicate function that detects which events are affected by a given channel. `hac:chpsim-event-may-block-deps-internal` is a primitive procedure that returns a structure listing dependencies, and `dependence-set-channels` selects the subset of channel dependencies by id number. The final result is a stream of event

indices that match the criteria. The following example demonstrates its use on input Program 3.1:

```
hacchpsimguile> (define ch (hac:parse-reference "Y"))
hacchpsimguile> (cdr ch)
1
hacchpsimguile> (define ch-event-strm
  (chpsim-find-events-involving-channel-id
    (cdr ch) all-static-events-stream))
; make a temporary associative list
hacchpsimguile> (define ch-astrm (stream-map
  (lambda (e) (cons (car e) #t)) ch-event-strm))
; sort into an ordered set
hacchpsimguile> (define ch-event-set (alist->rb-tree
  (stream->list ch-astrm) = <))
hacchpsimguile> (rb-tree/for-each-display-newline
  ch-event-set)
(1 . #t)
(2 . #t)
; Events 1 and 2 affect channel Y
```

3.4.1 Sharing and caching computed results

"Use the Force, Luke."

Obi-wan Kenobi, *Star Wars*

Many analyses start with the same set of queries. For instance, all analyses that examine branch selection statistics will start by identifying all branch events. Ideally, such common information should be shareable across similar analyses and never computed more than once, and also computed only when needed.

Fortunately, the Scheme language gives us the ability to *memoize* results of computations. The `delay` syntax and `force` procedure work together to accomplish delayed evaluation. In Scheme, `delay`-ing an expression makes a *promise* to evaluate it when it is called upon with `force`. We illustrate their operation with an example.

```
guile> (define (hello) (display "Hello!") (newline))
guile> (define y (begin (hello) (+ 1 2 3 4 5)))
Hello!
; non-delayed expressions are evaluated immediately
guile> (define x (delay (begin (hello) (+ 1 2 3 4 5))))
; does not evaluation expression, which would call hello
guile> x
#<promise #<procedure #f ()>>
guile> (force x)
Hello!
15
; evaluates expression, and memoizes result
guile> (force x)
15
; second call returns memoized result without re-evaluating
```

The delayed expression is not evaluated until it is first `force`d. The second call to `(force x)` simply returns the result that was memoized (saved) from the first call, without re-evaluating the delayed expression. This feature allows one to create *chains of dependent analyses*, with the benefit of computing intermediate results exactly once on demand, and re-using them.

For example, we can arrange graph edges as sorted adjacency lists for efficient lookup with the following structures.

`static-event-successors-map-delayed` [Variable]

> For efficient lookup, the successor-adjacency lists for the whole program event graph are available as a two-dimensional, sparse, ordered map. (Program E.4)

`static-event-predecessors-map-delayed` [Variable]

> The corresponding predecessor-adjacency lists of the event graph are also available as a two-dimensional sparse, ordered map. The predecessor map is the inverse of the successor map; each adjacency list

contains all events whose outgoing successor edges are incident upon an event node. These variables are memoized in a delay-force manner, so their values are only ever computed once per session and cached. (Program E.5)

`static-events-with-multiple-entries-delayed` [Variable]

Computed set of events with more than one predecessor. (Program E.7)

Loops. Loop head and tail pairs can be found statically by traversing event graphs. The forward and reverse maps are evaluated at the same time.

`static-loop-bound-events-delayed` [Variable]

This uses depth-first-search to identify events that complete loops. The result is a pair of ordered maps: loop heads are associated with loop tail events, and the corresponding reverse map. (Program E.10)

The individual loop head-to-tail and tail-to-head maps can be accessed using:

```
(define static-loop-head-events-delayed
  (delay (car (force
    static-loop-bound-events-delayed))))
(define static-loop-tail-events-delayed
  (delay (cdr (force
    static-loop-bound-events-delayed))))
```

With loop head and tail pairs pre-computed, one can query whether an event is a loop head or tail with a single lookup. (Most ordered lookup structures are implemented as red-black trees, whose interface procedures are listed in Appendix C.3.)

Program 3.3: `chpsim-event-loop-head?` procedure

```
(define (chpsim-event-loop-head? id)
; @var{id} is a static event index
  (rb-tree/lookup
    (force static-loop-head-events-delayed) id #f))
```

Program 3.4: `chpsim-event-loop-tail?` procedure

```
(define (chpsim-event-loop-tail? id)
  (rb-tree/lookup
    (force static-loop-tail-events-delayed) id #f))
```

Convergent branches. Analogously, branch head-tail pairs can be computed statically:

`static-branch-bound-events-delayed` [Variable]

This uses a depth-first traversal to compute the set of branch head-tail pairs, producing a forward map and a reverse map. (Program E.13)

The branch head-to-tail and tail-to-head maps are defined:

```
(define static-branch-head-tail-map-delayed
  (delay (car (force
    static-branch-bound-events-delayed))))
(define static-branch-tail-head-map-delayed
  (delay (cdr (force
    static-branch-bound-events-delayed))))
```

With branch head and tail pairs pre-computed, one can query whether an event is a branch tail with a lookup procedure:

```
(define (chpsim-event-branch-tail? id)
  (rb-tree/lookup (force
     static-branch-tail-head-map-delayed) id #f))
```

Concurrent sections. Concurrent forks and join pairs can also be evaluated statically.

`static-fork-join-events-delayed` [Variable]

This uses a depth-first traversal to identify fork-join event pairs. (Program E.14)

The complementary maps can be accessed as delayed variables:

```
; to reference the fork-to-join forward map
(define static-fork-join-map-delayed
  (delay (car(force
    static-fork-join-events-delayed))))
; to reference the join-to-fork reverse map
(define static-join-fork-map-delayed
  (delay (cdr (force
    static-fork-join-events-delayed))))
```

Loops, branches, and forks are just a few examples of statically computable event-graph information that may be frequently sought. The delayed evaluation computes the lookup structures only when queried, and only once even when called from different contexts. The important idea is that the interface to primitive procedures and relevant data structures allows one to derive static information in the form of procedures and delayed structures.

3.4.2 Using shared results

One benefit of being memoizing computed results is that new queries procedures can easily take advantage of delayed expression evaluation. We provide a function for general depth-first graph traversal (DFS),
`static-events-depth-first-walk-predicated`, listed as Program E.8. This procedure depends on the successor map, `static-event-successors-map-delayed`, and memoizes it when invoked for the first time. The procedure argument, *thunk*, expects a static event node index as an argument. The predicate argument, *pred?*, determines whether or not to visit the successors of an event.

`(static-events-depth-first-walk-predicated` ***thunk pred?***`)` [Procedure]

> Perform predicated depth-first traversal of the whole program event graph. (Program E.8)

The un-predicated depth-first traversal is simply defined as:

```
(define (static-events-depth-first-walk thunk)
  (static-events-depth-first-walk-predicated
    thunk (lambda (x) #t)))
```

These DFS procedures, however, are not properly tail recursive because of the visit stack bookkeeping in the tail call position. As a result, event graphs with long loops quickly run out of stack space. We also provide an iterative version of the DFS that requires constant stack space (for event graphs with very deep loops) in Program E.9.

To print out the sequence of visited event indices, one can call:

```
(static-events-depth-first-walk
  (lambda (n) (display n) (newline)))
```

Again, due to memoization, `static-event-successors-map-delayed`, will only be computed once and reused for all subsequent forced calls.

The next example filters out all events with exactly one successor and one predecessor, which is one way of identifying basic-blocks of control flow graphs:

```
(chpsim-filter-static-events-indexed
  (lambda (e) (and
    (= (rbtree/lookup (force
         static-event-predecessors-map-delayed)
      (static-event-node-index e) #f) 1)
    (= (hac:chpsim-event-successors
         (static-event-raw-entry e)) 1)))
  all-static-events-stream)
```

3.5 Trace analysis

A major contribution of our infrastructure is ability to analyze trace data in the Scheme environment, which allows interaction with trace data and convenient development of trace analyses. In this section, we present procedures for reading trace files and construct our first analysis routines.

3.5.1 Trace file content access

The first requirement for any trace analysis framework is to provide access to all trace file contents. Recall that the trace file contains two components: a history of all events, and a history of all values of variables. The `chpsim` trace API contains primitive procedures for accessing individual events in the history (Appendix F).

We provide three modes of access to the event history:

- *forward iterators*: advance forward in time one event
- *reverse iterators*: retreat backward in time one event
- *random access*: jump to any arbitrary event in history

Although these access modes are redundant (some may be defined in terms of others), they are implemented as different *handle* types to improve performance. By exploiting events' temporal locality, these specialized handle types can efficiently access large trace files on disk. Appendix F.1.1 describes the basic operations that open trace files in different modes, and the primitives that query the state of trace handles.

The most useful method for accessing arbitrary elements in the event history is `hac:lookup-trace-entry`, which uses a random-access trace handle. With the ability to access any element in the trace data, one can traverse entire histories through *stream* interfaces.

Event trace element. Each element in the event history is a tuple containing at least the following information: absolute index (ordinal number), timestamp of occurrence, static event index (refers to node in whole-program event graph), critical predecessor event. The procedures for accessing these fields are:

- `chpsim-trace-entry-index` – global event sequence number
- `chpsim-trace-entry-time` – event occurrence timestamp
- `chpsim-trace-entry-event` – unique event index
- `chpsim-trace-entry-critical` – critical predecessor (sequence number)

Their definitions are simply indexed references into Scheme lists (using `car-cdr` compositions).

State trace element. Each element in the value-change history is a tuple containing: the absolute index of the event that caused the value change, and the set of values changed with their new values. A *set* of values is required per entry because some events (namely, channel receives) may change multiple values atomically. Procedures for access value-change fields are listed in Appendix F.2.2. The event index is accessed using `chpsim-state-trace-entry-index`. The set of boolean variables changed per event can be accessed with `chpsim-state-trace-entry-bools`, and analogous accessors exist for integers and channels. Each variable set is a list of (index, value) pairs, where the indices correspond to global indices assigned when the simulation's variable state is allocated.

3.5.2 Trace file streaming

Streams are abstractions for both finite and infinite sequences of values. In Scheme, a stream is constructed from procedures using delayed evaluation (`delay`), where each element is not computed until it is actually referenced (by `force`). Guile memoizes delay-force pairs so references to previously evaluated stream elements quickly return the cached value (Section 3.4.1). Streams are also memory-efficient because they consume only as much memory as referenced, which is important for handling large trace files.

Stream interfaces to `chpsim`'s trace files mitigate the need to work directly on individual trace elements through a compiled library API. Appendices F.1.2 and F.2.1 list several stream-constructing procedures that operate on trace files. The most commonly used procedures that return streams are:

- `open-chpsim-trace-stream` views the trace of events forward in time

- `open-chpsim-trace-reverse-stream` views the event trace backwards
- `open-chpsim-state-trace-stream` views the history of value changes caused by all events in chronological order

These procedures are used in the vast majority of trace analyses.

3.5.3 Trace stream manipulation

Providing a trace analysis framework gives users the ability to construct arbitrary analyses without dealing with implementation details of the simulators and the trace files. Most importantly, a framework liberates users from the limitations of analyses developed solely by the authors of design tools. Manipulation of event and state streams can be demonstrated with a few examples. Basic stream procedures are described in Appendix C.4.

Program 3.5: Extract subset of event history on one particular event

```
; N is defined to a global event index
(define tr (open-chpsim-trace-stream "tracefile"))
(define result
  (stream-filter
    (lambda (e) (= (chpsim-trace-entry-event e) N))
  tr))
; result is a stream of occurrences of only event N
```

Narrowing the time window of event traces is useful for analyzing phases of the event history separately. The following examples select a subset of the event trace history by timestamp.

Program 3.6: Truncate a prefix of an event stream before a given time

```
; T is a time from which to start
(define tr (open-chpsim-trace-stream "tracefile"))
(define result
  (stream-start
    (lambda (e) (>= (chpsim-trace-entry-time e) T))
  tr))
```

Program 3.7: Truncate a suffix of an event stream after a given time

```
; T is a time at which to stop
(define tr (open-chpsim-trace-stream "tracefile"))
(define result
  (stream-stop
    (lambda (e) (<= (chpsim-trace-entry-time e) T))
  tr))
```

Program 3.8: Crop an event stream within a given time span

```
; T1 is a time from which to start, T2 is stop time
(define tr (open-chpsim-trace-stream "tracefile"))
(define result
  (stream-crop
    (lambda (e) (>= (chpsim-trace-entry-time e) T1))
    (lambda (e) (<= (chpsim-trace-entry-time e) T2))
  tr))
```

To select the corresponding subset of value changes in the same window of time, crop the state change stream using event occurrence indices.

Program 3.9: Crop an state-change stream within a given event span

```
; E1 is a start event, E2 is stop event
; E1 and E2 can come from chpsim-trace-entry-index
;   for an already cropped event stream
(define tr (open-chpsim-state-trace-stream "trace"))
(define result
  (stream-crop
    (lambda (c)
      (>= (chpsim-state-trace-entry-index c) E1))
    (lambda (c)
      (<= (chpsim-state-trace-entry-index c) E2))
  tr))
```

One can also filter events in the state-change stream by variable, which is accomplished by `chpsim-state-trace-filter-reference`, Program F.1. The resulting stream is a subset of the state-change stream, which can also contain information about other variables that changed at the same events. To strip away information about unwanted variables, the resulting stream can be restructured with `chpsim-state-trace-focus-reference`, Program F.2. If one is interested in only the sequence of values for the referenced variable, the variable

index field (which is now the same for all entries) can be stripped away using `chpsim-state-trace-single-reference-values`, Program F.3.

These example emphasize the ease with which data streams of `chpsim` event traces can be manipulated and restructured in the Scheme environment. Convenient access to trace data greatly simplifies development of new analysis procedures.

3.5.4 Combining static and trace analyses

With both static structure (Section 3.3) about the CHP program and run-time event trace histories available, one can easily construct analyses that leverage static and dynamic information. Two simple examples of such analyses are branch histograms and loop histograms.

Program F.7 lists the procedure, `make-select-branch-histogram`, for counting the frequency of successor events taken per selection statement. The procedure is outlined as follows:

1. `(chpsim-assoc-event-successors ...)` constructs an adjacency list of the program event graph using only select events (cached)
2. `(chpsim-successor-lists->histogram ...)` initializes a histogram
3. `sorted-asoc-pred` is a reverse map of predecessors constructed from the forward adjacency list from step 1. This reverse map speeds up predecessor lookup.
4. `count-selects` is the counting procedure that traverses all event stream elements to update the histogram
5. return populated histogram, `ll-histo`

The procedure for counting occurrences of loops, `make-loop-histogram`, is listed as Program F.8. The procedure is outlined as follows:

1. `(force static-loop-head-events-delayed)` caches the static set of all loop events in the program

2. each loop-head event initializes a slot in the histogram
3. every occurrence of a loop-head event in the event stream increments its counter
4. return final loop-count histogram

These examples demonstrate how easily one can collect useful trace statistics using static event queries and event streams in the functional programming environment.

3.6 Critical Path Analysis

One analysis that deserves more attention is critical path analysis because it lies at the core of many performance evaluations. The basic critical path procedures we describe in this section are listed in Appendix F.2.3.

3.6.1 Algorithm and implementation

The precise critical path can be deduced by querying each event for its last arriving predecessor event, and repeating for each critical event, progressing backwards through the event history.

1. let e = last event index
2. while $valid(e)$
3. record e
4. $e = lastpred(e)$

Since event criticality is frequently sought, it was deemed worthwhile to track and record critical events as they occur in the simulator event trace. The alternative would have been to reconstruct the critical path by examining all candidate predecessor events, which is slower to compute.

With critical events already tracked, the implementation of the critical path algorithm is very simple, listed in Program F.4. The procedure takes a random-access event trace handle, and with each iteration, seeks backwards in the trace handle to the critical event of the current event. The algorithm terminates at the first event (by detecting a self-reference). The case studies in Section 5.1 contain some examples of critical paths output by this procedure.

3.6.2 Critical path statistics

A critical path through a long event trace can be an overwhelming amount of information to grasp at once. We describe a few common ways of aggregating critical path information into statistics. One can quickly examine the frequencies of events found along the critical path.

1. for each event index i on critical path
2. ++event-counter[i]

The most frequent events are likely targets for optimizations. (The Scheme procedure for constructing a critical event histogram from the critical path is left as an exercise to the reader.) A simple first-order histogram does not convey any information about the sequences of critical events. Counting occurrences of adjacent event pairs along the critical path captures more sequencing information:

1. for each successive critical event-pair (e_i, e_{i+1}) on critical path
2. ++event-counter[e_i][e_{i+1}]

Program F.6 returns a sparse matrix where the $(i, j)^{th}$ values count the number of occurrences where event i was critical to event j. Such higher-order histograms are more effective at capturing correlations in event sequences.

In decomposed parallel programs, many critical paths will trace through channels connecting processes. Occurrences of channel communication events on the

critical path are meaningful to slack matching and pipeline optimization. Paired send and receive events on the critical path indicate whether the sender or receiver is the bottleneck. Using the trace information with critical path routines, we can construct an analysis to report which channels are critical and whether the sender or receiver side is more critical (Appendix F.2.6). Starting with a critical path and one channel of interest, the procedure flow is filter, fold, and count:

1. Filter-include all critical events that involve a given channel. (Program F.9, `make-critical-channel-event-pairs-list`)
2. Fold: pair together atomic send-receive event pairs. (also Program F.9) Not every communication event will necessarily be paired with its counterpart on the critical path.
3. Among the remaining paired channel events, count the number of occurrences of the send or receive event being more critical.
   (Program F.11, `count-send-receive-criticality`)

Since critical send-receive *pairs* are the only indicators that a critical path has crossed process boundaries[11], the presence of send-receive pairs on the critical path indicates that a design is limited by forward or backward latency through multiple processes, whereas the absence of such pairs indicates that a single process is a throughput bottleneck.

Another common critical path statistic simply counts how often critical events belong to various processes. Each static event index can be traced back to its owner process index, which is done by the `make-critical-process-histogram` procedure, listed as Program F.13. The resulting histogram provides only a rough approximation of the importance of critical processes because the event counts do not account for the amount of time spent in each process. Nevertheless, the result gives designers an idea of where to focus optimization efforts.

We use these procedures widely in many examples in Chapters 4 and 5 to determine how best to apply certain program transformations.

[11] except for shared variables, which are less common

3.6.3 Slack time computation

A more detailed look at critical paths can reveal the potential benefits of optimization. *Slack time* is the time difference between the most critical predecessor and the second most critical predecessor (if there is one). In other words, it measures the potential speedup available by optimizing only a critical event before becoming limited by the next critical path. Equivalently, it is also the delay increase that a non-critical path can withstand (e.g., from pessimization or trading-off performance) before becoming the new critical path and possibly degrading performance.

Since slack time is not computed and recorded on-the-fly, it must be measured from execution traces. Slack time is only applicable to events with more than one *necessary* predecessor. The recipe for computing slack time for each event is as follows:

1. for each event with multiple necessary predecessors
2. record event time of each predecessor
3. sort predecessors by event time
4. slack time is difference between two most recent predecessors

Events that can have multiple necessary predecessors include channel sends and receives and concurrent joins. (Branch joins only require one predecessor to proceed.) Wait statements and blocking deterministic selections (those that lack an else-clause) can have multiple predecessors, depending on the guard expressions. For example, $[a \wedge b]$ could be waiting for two separate events that set a and b true. Identifying necessary predecessors involving guard variables requires examining the values of the variables and their changes, from the variable history component of the trace file.

3.6.4 Critical path sensitivity

Slack time for each event on the critical path can indicate the amount of performance one might expect to gain by optimizing each event. However, slack time is only an approximation of the *sensitivity* of overall performance to each event's delay in the whole program. Performance sensitivity can help prioritize avenues of design space exploration, which is one of the goals of our analysis framework.

There is a significant amount of previous work on approaches to quantifying critical path sensitivity from parallel program analysis (software). In software, one asks how each segment of code affects the performance of a parallel program, whereas in hardware, one asks how each subcircuit impacts the system performance. The approaches described here mostly apply to measuring performance sensitivity on instrumented parallel programs. Since we are simulating execution, we have the liberty to alter event delays to mimic these methods.

S-Check (for sensitivity check) is an analysis tool that empirically determines where parallel program bottlenecks are by automatically inserting artificial delays at various program points and measuring its impact on performance [39, 61, 62]. Inserting artificial delays in the absence of nondeterminism guarantees identical intermediate results. This approach can be useful when one can afford (time and storage) to re-run the CHP simulation for each program point with altered delay.

Logical Zeroing (LZ) is a method for estimating the potential improvement in accelerating a part of a parallel program [49]. The improvement calculated by LZ is only an approximation because assigning a zero delay to part of a parallel program may cause events both on and off of the critical path to be reordered. True Zeroing (TZ) is an experimental technique that measures the *actual* potential speedup for a part of the program by replacing executed code with precomputed results (for correctness) [28]. True-zeroing was used to evaluate various parallel

program metrics used in estimating performance sensitivity. This approach is easy to support in our simulator by changing the delay of a particular event to zero.

3.6.5 Near-critical paths

Alexander, et al. present efficient algorithms for computing *near-critical* paths, given a program activity graph (from timestamp-annotated traces) with known slack times [2, 3]. An alternative to the above zeroing-based sensitivity analyses, near-critical paths help estimating the expected speedup (or slowdown) of changing the delay of individual events in the static event graph, which can help prioritize optimizations to explore. An activity graph with very low slack times exhibits multiple paths with similar delays. Since all events have fixed delays in our simulation model, we can easily reconstruct a time-annotated program activity graph for near-critical path analysis.

We can construct *all* near-critical paths from a simulation trace by extending the original critical path algorithm to use a worklist with a slack time budget. The following algorithm answers the question: what are all event paths that occur within a given slack time from the critical path?

1. while worklist has (event,budget) pairs (e, b)
2. let c be the critical predecessor of e
3. for all necessary predecessor events p at time $t(p)$
4. let $s = t(c) - t(p)$ be the slack time
5. if slack time $s < b$
6. add $(p, b - s)$ to worklist (remaining slack)
7. end for
8. end while

The input to the algorithm is a terminal event e_f, and an allotted slack budget b_0, which form the initial pair in the worklist. The algorithm always includes the

critical path; an initial slack budget of 0 is equivalent to finding only the critical path(s), because all events on the critical path have zero slack. This algorithm finds all event paths that lead to the critical path within the given slack budget. If the resulting directed-acyclic graph of events are annotated with their slack budget values $b(e)$, then the *maximum* additional delay that event e could afford before becoming the critical path is $b_0 - b(e)$ because any increase in delay reduces the available slack budget. It is possible for events to be visited multiple times from different re-convergent near-critical paths. To resolve this, the event should be re-evaluated using the minimum of all remaining slack budgets (over all incident paths) to capture the worst-case impact of increasing delay.

The set of near-critical paths presents even more information which may overwhelm the user, so one will often collect aggregate statistics about near-critical paths before scrutinizing the details. When prioritizing optimizations and transformations on the basis of potential improvement, it can be important to consider the available slack times in addition to frequency of occurrence along the critical path.

3.7 Putting it all together

This chapter has presented the infrastructure for analyzing programs and developing custom analyses. CHP is the high-level source language used to describe concurrent programs. We have provided a compiler and CHP simulator that can produce event traces from run-time execution. Rather than provide analysis routines and trace access methods through a compiled programming interface, we make all static and run-time information accessible through primitive procedures in an interactive Scheme environment. The Scheme environment makes it very easy to manipulate data and develop new analysis procedures with little hassle.

The examples in this chapter demonstrate how effortlessly procedures can be written and deployed. An added benefit of an interactive analysis environment is that users can select the analyses of interest depending on the data observed.

In the next chapters, we describe how run-time analyses developed in our framework can aid in selecting and exploring optimizing program transformations.

CHAPTER 4

APPLICATIONS OF ANALYSES TO TRANSFORMATIONS

A program should be light and agile, its subroutines connected like a string of pearls. The spirit and intent of the program should be retained throughout. There should be neither too little or too much, neither needless loops nor useless variables, neither lack of structure nor overwhelming rigidity.

A program should follow the 'Law of Least Astonishment'. What is this law? It is simply that the program should always respond to the user in the way that astonishes him least.

A program, no matter how complex, should act as a single unit. The program should be directed by the logic within rather than by outward appearances.

If the program fails in these requirements, it will be in a state of disorder and confusion. The only way to correct this is to rewrite the program.

The Tao of Programming

The ultimate goal of our asynchronous CAD tools is to be able to automatically and efficiently explore high-level transformations of parallel programs, which lead to optimized circuit synthesis. A crucial step towards that goal is to provide a framework for analyzing the performance of asynchronous circuits at a high level of abstraction. In this chapter, we discuss various program transformations used to optimize parallel programs. The sample of transformations is far from exhaustive, however, the point is to show how our trace analysis framework can aid in deciding how best to apply an arsenal of program transformations.

Before a program can be rewritten, static program analyses are required to determine how one can locally rewrite components without altering the outcome of the program. *Semantic-preserving* transformations are the heart of optimizations of software and hardware. Local (non-visible) results are permitted to change as long as the visible outcome is consistent with the original program. Techniques for static analyses and program rewriting are outside the scope of this dissertation,

however, we provide examples where the applicability of optimizing transformations is not deducible from static analysis alone.

Asynchronous VLSI liberates circuit designers from minding timing constraints on signals; a purely event-driven hardware abstraction (and programming abstraction) lets designers and synthesizers focus on preserving the *sequences* of values observed at process interfaces for correctness, independent of timing and performance. The synchronous design methodology does not afford this freedom to decouple timing from functional correctness, and is thus harder to design and decompose modularly, and difficult to verify formally. We discuss this point in more detail in Section 4.2.

4.1 Parallel Decomposition

The whole is more than the sum of its parts.

Aristotle, *Metaphysics*

The sum of the parts is greater than the whole.

overheard in a ceramic repair shop

The most common transformation used in concurrent hardware is parallel decomposition, where longer sequential loops are broken down into semantically-equivalent sets of shorter, and explicitly concurrent loops. Monolithically sequential functional specifications of large systems can be progressively decomposed into smaller and simpler concurrent processes. One difference between parallel software and parallel hardware is that hardware exists as repetitive processes that respond to their inputs *at all times*; the existence of circuits is not lexically scoped, nor is their operation determined by call-sites. (For readers more familiar with parallel software, asynchronous hardware design is analogous to message-passing multithreading, or concurrent processes communicating over channels or sockets.)

Tradeoffs. Since decomposition results in more processes and communication channels, finely decomposed processes will be naturally more pipelined. Smaller pipelined processes are easier to implement and synthesize into circuits. The added parallelism from decomposition can result in increased throughput, but comes at the cost of area overhead from circuits used to communicate over channels. Increased parallelism can contribute to an increase in activity density and power consumption. Less pipelined (coarser-grained) designs may be more suitable under area-limiting or power-limiting constraints. In designs where the activity is non-trivially input-dependent, trace analysis can help identify which subcircuits are worth decomposing.

Techniques. There many approaches to process decomposition using static program analysis. *Projection* is a method that divides program variables and actions on variables (such as channel communications) into non-conflicting sets of producer-consumer communications [41]. Many techniques from conventional software compilers are also highly applicable to decomposition. Static-single assignment (SSA) form is useful for identifying the lifetime of each variable's definition, and explicitly merging multiple definitions (using ϕ nodes). By separating each definition into its own def-use chains, it is easier to separate different definitions of the same variable into different communicating processes. 'Static tokens' is a more restrictive variation of SSA that has been used to decompose sequential descriptions into fine-grain asynchronous primitive processes, and has been used to map CHP programs onto asynchronous FPGAs [68]. Basic dataflow analysis is useful in determining the lifetime of every variable definition, which determines where interprocess channels are needed in decomposition [77, 78].

Composition. When taken to the extreme, decomposition can produce fine-grain, overly decomposed processes that incur high communication and area over-

heads. It is natural to ask: when and where is it beneficial to coalesce several communicating processes into fewer, unpipelined processes? Explicit channels and process communication interfaces make it very easy to re-compose parallel process back into sequential processes. Since the number of combinations of compositions is potentially exponential with the number of components, finding an efficient *composition* can be an intractable without examining the utilization and criticality of each component. For example, combinations of compositions could be greatly pruned by eliminating components that appear on the critical path. The information one can gather from trace analysis can be used to guide both decomposition and recomposition of parallel processes.

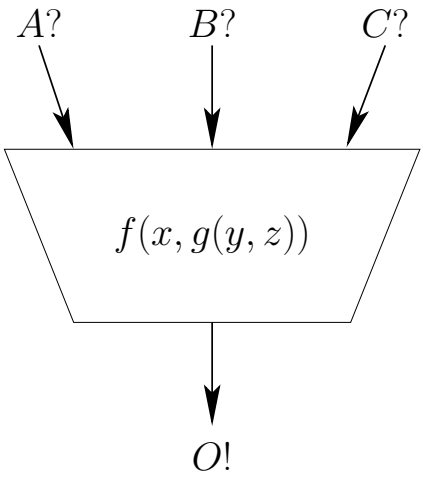


Figure 4.1: Unpipelined expression computation process

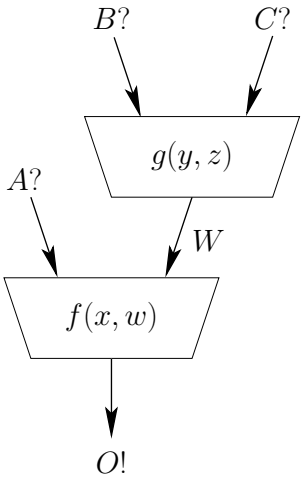


Figure 4.2: Pipelined expression computation process

Functions and expressions. Another context where decomposition can be applied is in functions and expressions. Consider the following feed-forward computation (Figure 4.1):

$*[A?(x), B?(y), C?(z); O!(f(x, g(y, z)))]$

Since CHP's expressions are not inherently pipelined, $f(x, g(y, z))$ (in its literal interpretation) could be synthesized as a monolithic, unpipelined function block. If the bottleneck lies in the evaluation of $f(x, g(y, z))$, and the process's throughput

(rate of repetition) is performance-critical, then pipelining the $O!(...)$ action would improve throughput by decoupling the computation from the communication on O. The process can be rewritten to explicit use an intermediate value w:

$$*[A?(x), B?(y), C?(z); w := g(y,z); O!(f(x,w))]$$

The assignment of w can be decomposed into a send-receive pair on channel W between concurrent processes (Figure 4.2):

$$*[B?(y), C?(z); W!(g(y,z))]$$
$$\| *[A?(x), W?(w); O!(f(x,w))]$$

Each new process evaluates simpler expressions, and is expected to achieve greater throughput than the original process. By decoupling the producer and consumer of w, the producer process can concurrently begin the following iteration while the consumer is computing. Pipelining, however, incurs forward latency in communication over channel W. It only makes sense to pipeline the original computation if its throughput is limiting the overall performance, not when the latency of result is critical. Critical path analysis can help categorize processes as throughput-critical or latency-critical, which can help human (or machine) rewriting of the high-level parallel program description.

One of several ways to determine the propriety of pipelining through decomposition is to ask *how* the channels of this process (A, B, C, O, W) appear on the critical path, using channel criticality procedures from Section 3.6.2. The result of `make-critical-event-pairs-list` immediately indicates whether the problem is isolated to a single process or crosses multiple processes: if the result lacks send-receive *pairs* on the critical path, then the performance is limited to the throughput of a single process because the critical path never crosses process boundaries. In the cases where channel event pairs do appear: if a channel is *repeatedly* sender-critical, then performance is limited by forward latency, a receiver-critical channel is limited by backward latency.

Process decomposition is an essential transformation from which sequential programs first gain large speedups due to parallelization. Static program analyses can help partition large processes into communicating subprocesses. When applied to the extreme, decomposition results in finely pipelined, simple processes operating concurrently. A designer who concerns herself with the tradeoffs between area, energy, and performance can benefit from identifying components with favorable tradeoffs using our trace analysis framework. A profile-guided exploration of decompositions (or compositions) can drastically reduce the space that would otherwise be intractable.

4.2 Pipelining and Slack Matching

Process decomposition naturally adds pipelining to a concurrent program because the producer and consumer of communicated channel values can operate decoupled from each other. Pipelining often leads to increased performance because the resulting decoupled processes have shorter loops, and thus higher achievable throughput. A parallel program can also be pipelined by adding buffers (FIFOs) on the communication channels between processes, which increase the slack of channels [75]. *Static slack* is defined as the maximum difference between the number of communications (tokens) observed on the ends of a channel, in other words, the token capacity of a channel. To better understand why slack matching is only relevant to asynchronous VLSI, we summarize the differences between synchronous pipelining and asynchronous pipelining.

Synchronous vs. asynchronous pipelining. Changing pipelining in asynchronous circuits without affecting its functional correctness is possible because correctness is only defined by the sequence of values observed, not their timing; there is no notion of expecting signals at clock edges. Pipelining a synchronous

design results in additional clock cycles latency, which alters its observable interface. The problem of pipelining a synchronous design, known as *retiming*, involves relocating clocked register boundaries, possibly adding or removing registers on paths, and can be very invasive to change. (A specification of the logic between clocked latches is called Register Transfer Logic, or RTL.) Synchronous retiming becomes non-trivial as soon as there are cycles of clocked paths created by internal feedback; changing the amount of pipelining on cycles causes a visible change in functionality! A cyclic clocked path with N registers computes a different result than the same cyclic path and logic re-timed to use M registers! A synchronous cyclic path with N registers invariably holds N values in the loop.

Asynchronous pipelining, however, can deepen pipelines *without adding placeholders for values*, in other words, the physical pipelining (number of FIFO buffers) is *independent* of the logical pipelining (number of value places). (One can, however, add 'initial-token' buffers to increase the number of value places to asynchronous pipelines.) This property allows channels (on cyclic and acyclic paths alike) to be pipelined with arbitrary depth without changing the functionality of the whole program for a wide class of designs, *slack elastic* designs [42]. Arbitration-free asynchronous designs without data races fall into the category of slack elastic designs. Some designs that use arbitration can be slack elastic. Asynchronous designs can also exhibit local slack elasticity in parts of the entire program.

Pipeline dynamics is the general study of asynchronous pipeline performance with respect its structure: the number of buffers, the number of tokens in-flight, and the latencies through the buffers. Asynchronous pipeline performance is well understood for pipelines under steady-state operation [38, 75]. To summarize, the analytic solutions for asynchronous pipeline performance are computable as min and max expressions of the canonical sources of performance limitations:

- forward latency limiting (token-limited)

- backward latency limited (bubble-limited)
- cycle limiting (self-limiting or handshake-limited)

Other authors have proposed linear programming solutions to more generalized pipeline topologies [57]. Since adding buffers increases the energy consumed per token, one can also optimize a design for energy-efficiency by removing buffers from the throughput-optimal solution [67]. The limitation of all of these slack matching methods is that they are restricted to steady-state operation.

In practice, pipelined designs do not always exhibit simple steady-state behavior. For example, transient pipeline behavior can arise from data dependence, or pipeline hysteresis (state-holding), or even loops with varying number of tokens in-flight. It will not always be possible to find an analytic or numerical solution for every situation. Complicated parallel program behavior can be better understood through simulation and detailed execution analysis, which is the role of our trace analysis framework.

4.2.1 Intuition from criticality

Trace analysis can also help designers less familiar with asynchronous circuits understand pipeline dynamics using the principle of criticality. Consider a typical CHP program excerpt with channel communication actions:

$$*[...; X_i!(x); ...]$$
$$\| *[...; X_o?(z); ...]$$

where X_i and X_o are the respective input and output of a FIFO channel X. (The partial event graph resembles Figure 3.5, except that channel C is replaced with a FIFO.) The send event on channel X_i must wait for two preconditions before it executes: the immediate predecessor event has completed, and that the receiving end ($X_i?$) has been reached, and is ready to receive. Likewise the receive event

on X_o waits for its predecessors to complete, and for the sender side to have data ready to send ($X_o!$). If critical path statistics (Section 3.6) reveal that on the sender side, the predecessor is more critical, then performance is limited by either the forward latency through X or the repetition rate of the sender process. In this case, the receiver process will also find that the sender is more critical than the predecessor of $X_o?()$. If forward latency through a channel is critical, one can reduce forward latency by reducing the number of buffers of the FIFO on the channel. On the other hand, if the sender process finds that the receiver is more critical (the send event is usually waiting for the receiver to be ready), then performance can be improved by adding more buffering on channel X, or improving the receiving process's throughput. Additional static slack on channel X would further decouple the sender and receiver, allowing the sender to proceed further when the receiver is congested. If the throughput is limited by the repetition rate of either the sender or receiver process, then changing the amount of buffering on X will not yield any speedup.

4.2.2 Token ring examples

No discourse on slack matching would be complete without referring to the token ring, an asynchronous FIFO connected in a closed loop. We use token rings to demonstrate critical path analyses on trace files. Suppose we have a buffer whose forward latency is 1 time unit, and backward latency is 7 units, and thus has a cycle time of 8 units (denoted as a $(1+7)$-buffer, for brevity). We connect several such buffers in a ring with one element that contains an initial token.

A whole program event graph of a slack-6 ring is shown in Figure 4.3. (The event graph legend is summarized in Figure 3.2.) In this figure, each buffer is drawn as a separate process (shown as rectangles), where each process contains a

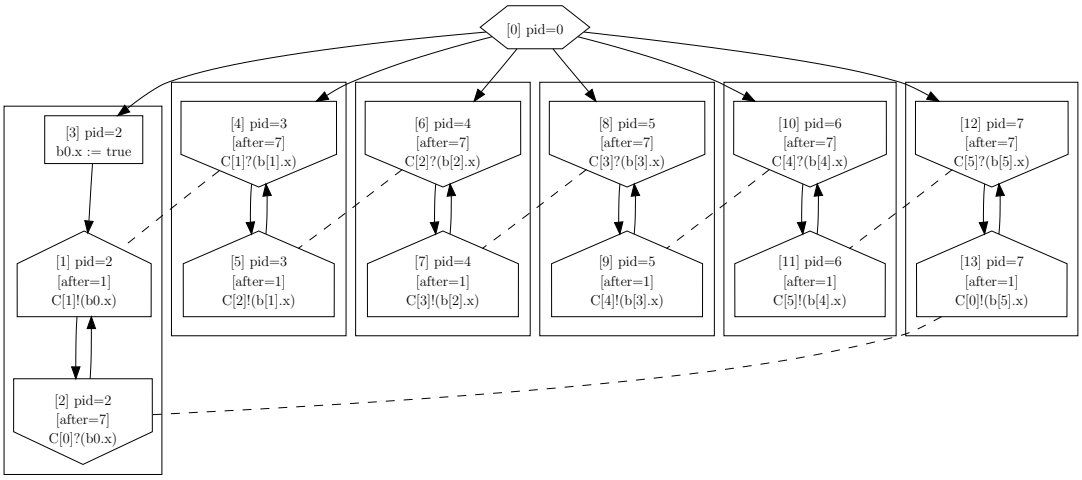


Figure 4.3: Whole program event graph of a token ring

simple event subgraph of a send and receive event in alternation. The initial token buffer (left) sends a token before receiving; there is exactly one token in the ring at all times. The dotted edges between events in different processes represent the channels over which values are sent. The send and receive events at the endpoints of a channel execute simultaneously and atomically, as described in Section 3.2.2, Figure 3.5.

After simulating and producing a trace, we examine the critical paths through token ring. Table 4.1 is an excerpt of the critical path through the simulation trace of this token ring. The 'index' column is the event number in the trace file, the 'time' is the time of event occurrence, 'event' is the index of the *static* event from the whole program event graph, and 'crit' is the index of the critical predecessor event, the last predecessor to unblock this event. The static event numbers correspond to those in Figure 5.2. Critical send-receive pairs are listed as paired rows. For every send-receive pair on the critical path, the event with the lower event sequence index (column 1) is the more critical of the pair, i.e. the program point was reached later than its counterpart with the higher index. For example, traced event index 182 (send event 4, in process 3) was more critical

Table 4.1: Critical path through a token ring, whose performance is limited by the buffers' cycle time. Send-receive event pairs have been grouped together.

index	time	event	crit.
.....			
183	137.0	1	182
182	137.0	4	173
173	130.0	5	172
172	130.0	6	163
163	123.0	7	162
162	123.0	8	153
153	116.0	9	152
152	116.0	10	143
143	109.0	11	142
142	109.0	12	133
133	102.0	13	132
132	102.0	2	123
123	95.0	1	122
122	95.0	4	113
113	88.0	5	112
112	88.0	6	103
103	81.0	7	102
102	81.0	8	93
93	74.0	9	92
92	74.0	10	83
.....			

Table 4.2: Critical path through a token ring, whose performance is limited by the buffers' forward latency. Send-receive event pairs have been grouped together.

index	time	event	crit.
.....			
85	94.0	2	84
84	94.0	13	83
83	92.0	12	82
82	92.0	11	81
81	90.0	10	80
80	90.0	9	79
79	88.0	8	78
78	88.0	7	77
77	86.0	6	76
76	86.0	5	75
75	84.0	4	74
74	84.0	1	73
73	82.0	2	72
72	82.0	13	71
71	80.0	12	70
70	80.0	11	69
69	78.0	10	68
68	78.0	9	67
67	76.0	8	66
66	76.0	7	65
.....			

than its corresponding receive (event 1, in process 2). As we follow the critical path backwards in time, we observe that each send-receive pair points to the *receiver* side as the critical event. The receive action in each buffer is the limiting factor because its delay of 7 units is longer than the total forward latency around the ring, 6 units; the cycle time of each buffer process limits the throughput of the token ring. The critical path analysis corroborates our expectations from pipeline dynamics.

The above analysis is captured by the `channel-send-receive-criticality` procedure (Program F.12), whose result is pair of counters for the number of occurrences of sender-criticality and receiver-criticality[1].

```
; 'crit' is already the critical path stream
; "ring.M[0..N]" are the channels in the token ring
hacchpsimguile> (channel-send-receive-criticality
  crit "ring.M[0]")
(1 . 3)
hacchpsimguile> (channel-send-receive-criticality
  crit "ring.M[3]")
(0 . 3)
```

The number of receiver-critical occurrences (3) always exceeds the number of sender-critical occurrences. The analysis finds that the ring of 6 $(1 + 7)$-buffers is limited by backward latency, which indicates to the designer that more slack or improved backward latency would improve performance.

Now consider the same token ring of buffers with different forward and backward latencies, $(2 + 6)$-buffers, with the same cycle time. (The event graph is unchanged from Figure 4.3.) An excerpt of the new critical path is listed in Table 4.2. This time we observe that for every send-receive pair on the critical path, the *sender* was always the more critical event (listed with lower index in each paired row). This concurs with pipeline dynamics principles: the cycle time is limited by the total forward latency around the token ring, which is now 12 units (6×2).

```
; 'crit' is already the critical path stream
; "ring.M[0..N]" are the channels in the token ring
hacchpsimguile> (channel-send-receive-criticality
  crit "ring.M[0]")
(9 . 0)
hacchpsimguile> (channel-send-receive-criticality
  crit "ring.M[3]")
(8 . 0)
```

[1]Counts below 2 are usually attributed to transient behavior from the beginning of simulation or the tail end of the critical path.

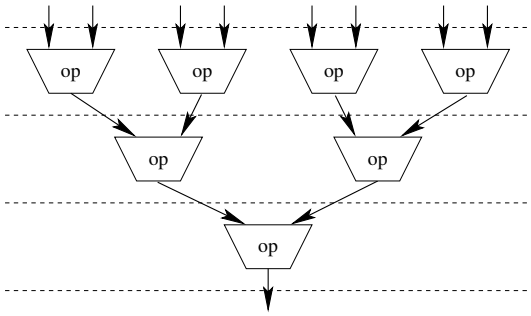


Figure 4.4: A balanced computation tree is suitable when inputs arrive close in time.

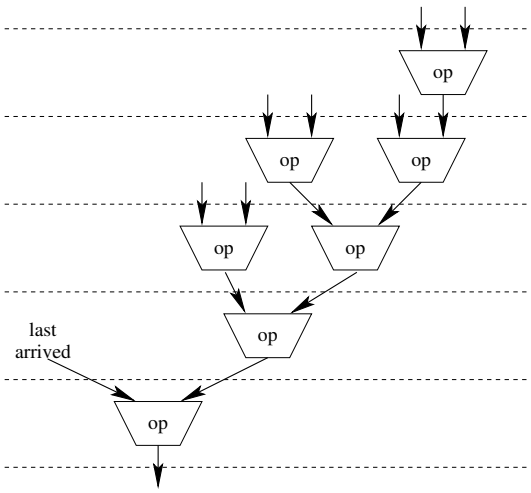


Figure 4.5: An unbalanced computation tree is suitable when the last input arrives much later than the rest.

This time, the same analysis finds that the sender is always more critical (8 and 9 times), which indicates to the designer that performance can be gained by reducing slack, or improving the forward latency of the buffers.

The example in this section demonstrates how basic pipeline dynamics are corroborated by critical path analysis. The critical path analysis procedures in our trace analysis framework can serve as a basis for exploring slack matching in arbitrarily complicated asynchronous circuits and parallel programs. Slack in an asynchronous design can be optimized without requiring a specialized framework for analyzing pipeline dynamics. Section 5.1 demonstrates how critical path analysis leads to the same conclusion as static pipeline analysis for a pipelined computation loop operating in a steady state.

4.3 Subexpression Scheduling

In Section 4.1, we described how expressions can be explicitly pipelined in CHP by communicating intermediate results on channels. Critical path statistics can in-

dicate whether decomposition (or composition) may improve performance. Other opportunities for optimizations can come from functions and expressions that can be evaluated in different orders, such as associative operations, often found in reduction computations. Common associative operations include addition, multiplication, bitwise- and logical- AND, inclusive-OR, exclusive-OR, minimum, maximum, least common multiple, greatest common denominator. (These all happen to be commutative, but commutativity is not required.)

Given an associative expression: $OP(a, b, c, d, ...)$ with varying arrival times of its inputs, what is the optimal tree-decomposition that minimizes the delay of the final result? To answer this, one considers the relative arrival times of the inputs with respect to when each output is produced. Intuitively, the last arriving input should be scheduled closest to the final result of the evaluation to allow evaluation of independent subexpressions as early as possible. Using our trace query framework, one can construct the following analysis:

1. for expressions o of the form $OP(...)$
2. note time t when o is completely evaluated
3. for all operand variables v_i used in expression o
4. find the event a that necessarily produces v_i (receive or assignment)
5. note the time of the event, $t(a)$
6. identify the "last arrived" variable as critical

Suppose we compute the result using only binary (2-input) function units. If N inputs arrive simultaneously, a balanced reduction tree results in the minimal delay in evaluating the result (Figure 4.4). However, in the extreme case where the last of N inputs arrives much later than the others, the optimal scheduling will evaluate as much as possible before the last input arrives with unbalanced structures (Figure 4.5). The slack times relative to the last arriving input and the latency per stage of computation will determine the optimal shape of the tree-decomposition. The problem is also generalizable to heterogeneous expressions

involving different operators each with different latencies, where the scheduling freedom is still limited to only associative operators.

To assess how an expression tree should be restructured to minimize average latency, one should focus on occurrences of the input channels (at the leaves of the expressions) in the critical path. In a forward-latency limited scenario, each occurrence of the final output channel on the critical path will also be preceded by one of the input channels on the critical path. In any given configuration, the input channel that appears the most often on the critical path is a likely candidate for 'pushing' closer to the root of the computation. Simple queries on the frequencies of occurrences of a set of channels on the critical path (using `make-critical-channel-event-pairs-list`, Program F.9, for example) will find good candidates for restructuring.

Subexpression scheduling is just one application of general slack time analysis (from Section 3.6.3), where a group of events are prerequisites to another event, whose dependency graph is tree-like in form. In this particular instance, the dependencies represent intermediate results of a large expression. Run-time critical path and slack time statistics can improve circuit synthesis by scheduling the frequently critical paths more aggressively. Subexpression scheduling is applicable to mapping concurrent computations onto programmable devices such as FPGAs. Particularly for asynchronous FPGAs, the place-and-route phase has potential to reduce significant latency on forward paths, given the knowledge of critical paths discovered from simulation.

Instruction scheduling in compilers. Software compilers also take advantage of knowing the different latencies of machine instructions. Back-end assembly code emitters use some model of the machine pipeline, and attempt to schedule instructions in an order that minimizes pipeline stalls (wasted idle cycles). Code

generators have the added constraint that machine code can only be read linearly in an instruction stream, though modern superscalar architectures can issue multiple instructions per cycle. The compiler's back-end optimization problem can be summarized: given availability times of inputs to a block of computation, find a static instruction scheduling that minimizes the latency of the final result(s).

In hardware design, the analogous challenge is to find a static restructuring of a computation that minimizes the typical latency to the final output. Unlike machine code generation, hardware design lacks the sequentializing constraint; independent operations may happen concurrently in an event-driven manner. We have shown how trace analysis can aid the optimization of a recurring pattern in circuit design, where multiple inputs are reduced to a single result through computation. In particular, the analysis procedures from Section 3.6.2 and Appendix F.2.6 are suitable for finding the most critical input to a computation, which is a prime candidate for restructuring. Besides expressions, other examples of convergent dependencies include synchronizations, and route merges. We will see the theme of criticality-driven restructuring again in Section 4.5.

4.4 Flow Control and Speculation

In this section, we look at opportunities to optimize around flow control constructs in parallel programs. Hardware and software optimization around control flow share many common ideas. Many software compilers feature optimizations that revolve around control flow. Branches have played a crucial role to machine code performance for many reasons: latency is incurred by having to wait for comparison outcomes before selecting an execution path, so branch delay slots were introduced in some ISAs to hide latency by scheduling branches before instructions that logically preceded the branch. Hardware branch prediction hides branch laten-

cies by speculatively executing down likely paths, but pays a performance penalty to undo the effects of misprediction. The greater the misprediction penalty, the more important it is to have high accuracy. Compilers have the added challenge of scheduling likely sequences of basic blocks together to improve instruction memory locality, and minimize disruption to the flow of instructions. Instruction predication alleviated overhead incurred on short branches by conditionally masking the effects of select instructions in longer basic blocks. Predication is usually accomplished through if-conversion on an intermediate representation such as SSA.

It is not always possible to infer optimal transformation policies from static program analysis alone. Wrongly applied transformations can degrade a program's performance! Program execution can be profiled to collect statistics on branches, which is featured in the `gprof` instrumenting profiler [22]. A compiler with profile statistics is much more capable of making more informed optimizations around flow control constructs. Many of these principles analogously adapt to asynchronous hardware design and optimization.

With access to entire simulation traces, our trace analysis framework provides an interface to construct arbitrary queries on the trace file offline. In Section 3.5.4, we outlined a procedure to collect branch statistics over an entire event trace, Procedure F.7. To focus on only branch events that occur on the critical path, one simply passes the critical path event stream (result of Program F.4) as the input. This is useful when one intends to transform only the most performance-critical branches in the parallel program.

Speculation. One novel application of branch statistics in compiler optimizations is *speculation* or speculative code motion, where some actions are taken before it is known whether or not their results are actually relevant or applicable. This can result in a speedup by computing results that are critically needed *before* waiting

for a conditional result to start the computation. Speculation incurs performance overhead when the result of speculative execution is thrown away, because the computation resources could have been spent elsewhere, so it is generally applied where the benefits outweigh the costs with high confidence. As mentioned earlier, branch prediction in microprocessors is one example of speculation in hardware. Confidence-based predictors allow a branch to be speculatively taken when it is highly probable that it will not be mispredicted.

Consider the following construct in CHP:

$$
\begin{array}{l}
*[P?(p), Q?(i); \\
\quad [p \rightarrow R!(i+1) [] \textbf{else} \rightarrow \textbf{skip}]; \\
\quad \ldots \\
\quad] \\
\| \\
*[R?(c); \ldots]
\end{array}
$$

A value is conditionally sent over channel R, and the receiving process waits for a value on R before continuing to execute. The first loop can be re-written to execute the then-clause *speculatively* and forward the predicate p to the consumer in the second loop to be able to correct misspeculation (since every producer must be matched to a consumer). The resulting equivalent program resembles the following:

$$
\begin{array}{l}
*[\{P?(p); B!(p)\}, \quad // \ copy-forward \ the \ predicate \\
\quad \{Q?(i); R!(i+1)\}; \ // \ B \ and \ R \ decoupled \\
\quad \ldots \\
\quad] \\
\| \\
*[R?(c); \\
\quad \ldots \quad // \ start \ some \ work \ speculatively \\
\quad B?(b); \\
\quad [b \rightarrow \textbf{skip} \\
\quad [] \textbf{else} \rightarrow \ldots \ // \ discard \ iteration \\
\quad] \\
\quad]
\end{array}
$$

With respect to energy, such "code motion" would be justifiable if the predicate p is usually true, so that few additional communications on R would be wasted, and the

early evaluation in the receiving process results in speedup. There is performance to be gained in the new version if the critical path frequently contains the path from Q to R; speculatively sending R allows the receiver to proceed earlier, even if its result is not always needed. To evaluate this tradeoff, one must profile the frequencies of the branches, and the slack times of speculative results on the branch:

1. for all conditional events (immediately dominated by a branch and post-dominated by the corresponding branch merge)
2. filter: actions that produce a result (e.g. send)
3. count number of occurrences on the critical path
4. optional: evaluate slack time w.r.t. sibling predecessors

In other words, conditional producers of values that are frequently found on the critical path are likely candidates to benefit from speculative execution. Large slack times and long latency operations on the conditional paths indicate opportunities to speedup from speculation. The results of the above analysis can serve as a starting point in finding opportunities to optimize parallel programs using speculation.

Another speculative transformation executes multiple branches concurrently and postpones result selection. In the following program, a function of two variables is selected based on a predicate b:

$*[B?(b);$
$\quad [b \rightarrow z := f(x,y)$
$\quad []\textbf{else} \rightarrow z := g(x,y)$
$\quad];$
$\quad Z!(z)$
$\;]$

If receiving on B is critical and f and/or g are slow operations, then performance is limited by sequencing the functions after receiving B. Here is an opportunity to pre-compute the results of f and/or g earlier, because neither expression depends

directly on b. After hoisting both $f()$ and $g()$ above the selection, the equivalent program becomes:

```
*[B?(b), s := f(x, y), t := g(x, y);  // concurrent, not blocked by B
  [b → z := s  []else → z := t ];  // one result is wasted
  Z!(z)
 ]
```

By evaluating both branches, the evaluations of $f()$ and $g()$ are no longer blocked by the predicate b. Again, speculation incurs an energy cost, where more results are computed than are actually used. The overall benefit of this transformation depends on the context of this process instance, and its impact on the whole system. If $B?(b)$ were not on the critical path, then there would be no benefit to speculatively computing results, only wasted energy.

The above examples could have also been postulated negatively: when does it pay to *de-speculate* work, postponing actions until it is certain that their results are needed? Such an analysis begins by asking how often values are defined without being used (e.g. def-use chains from dataflow).

1. find all variables v that may be *conditionally* unused
2. locate static events that produce them, $e = producer(v)$
3. identify paths from these events in which variables are *dead*
4. count occurrences in the event trace where paths taken leave variables dead
5. sort by: frequency-of-occurrence × energy-per-occurrence

This identifies opportunities to reduce energy by postponing computations until their results are guaranteed to be used. De-speculating transformations are worth exploring in energy-critical applications where performance does not matter. The following CHP program illustrates a scenario where de-speculation may be beneficial.

```
*[...;  a := h(x, y), P?(p);
   [p → z := g(a, x)
   []else → z := g(y, x)
   ];
   ...          // use z, but not a
 ]
```

Note that the path from the definition of a through the else clause results in a dead definition of a. If trace analysis found that a was frequently unused, then a compiler might be motivated to delay the computation of a onto paths where it is used.

```
*[...;  P?(p);
   [p → a := h(x, y); z := g(a, x)
   []else → z := g(y, x)
   ];
   ...
 ]
```

By computing a only on paths where it is needed, the transformed program now consumes less energy per iteration than the original. Whether or not the de-speculated program runs slower than the original depends entirely on criticality, as determined by trace analysis.

Considering scenarios where one may be interested in *selectively* moving code into or out of branches, or even introduce branches, one realizes that exploring the entire span of semantic-preserving speculation (or de-speculation) transformations is intractable. Static program analyses may identify abundant opportunities to apply speculation, but without profiled analysis, it can be very difficult to determine which sites are worth rewriting. Profiling a simulated execution trace for branching statistics and path criticality is a highly effective way of narrowing the scope of local transformations to consider. Our trace analysis framework gives users the ability to inspect execution details on events involving branches, which ultimately helps users (or compilers) make informed decisions to apply speculating

transformations.

4.5 Selection Restructuring

One of the general observations of circuit design is that larger circuits that compute more complex functions tend to be slower. The physical intuition behind this observation is that:

- circuits that compute more tend to have more transistors in series, thus reducing drive strength
- more transistors increases capacitive load (especially when sized for drive strength)
- longer wires spanning larger areas increase both capacitance and resistance.

This results in an interesting space of design tradeoffs for circuit designers in both the synchronous and asynchronous domains. For instance, at the transistor netlist level, designers are constantly faced with the decisions to split or combine logic gates. Should a large N-input function be divided into multiple levels? The mathematics of logical effort formulate such questions as delay minimization problems [65]. At the gate level, circuit designers are primarily interested in meeting target cycle times — the search for a solution is driven by "whatever it takes to meet the target."

The same size-performance tradeoffs exist at a higher level of abstraction in asynchronous circuits. Section 4.1 discussed a tradeoff between unpipelined and decomposed pipelined communicating processes. Larger unpipelined processes are typically slower and lower energy, and are appropriate when they are infrequently found on the critical path. From Section 4.3, the natural tree-like topology of many computations gives designers some freedom to decompose expressions into different structures, depending on the criticality of inputs.

Other common structures that can be optimized are selection structures in CHP. Selections can come from flow control statements such as branches and conditional loops, and implicit selections in indexed references, such as $X\texttt{[}i\texttt{]}?(x)$ or $y := x\texttt{[}i\texttt{]}$. Accesses to large memory arrays are a common target for optimization. Larger memories are slow due to long wires and high capacitive loads on word-lines and bit-lines. Memory arrays are often uniformly banked and partitioned to improve the physical properties for performance.

However, skewed access distributions of arrays can provide opportunities to further improve average-case performance by favoring common cases, as demonstrated in non-uniform access asynchronous register files [17, 18]. Asynchronous designs such as the Lutonium micro-controller and SNAP sensor-network processor have also featured multi-level datapaths that place frequently used functional units on a faster bus, while seldom used units accessed a second-level bus to reduce load on the fast bus [14, 30, 43]. (The basic principle is analogous with information theory, which explains how data can be compressed by encoding common words with shorter strings, such as Huffman coding.) Optimizing common cases at the expense of infrequent cases can result in a net improvement over uniform accesses. This paradigm is especially important to latency-tolerant and self-timed design styles such as asynchronous VLSI.

Selection structures can come in two flavors: one-to-many or many-to-one (many-to-many can be a composition of these). Writing to an element of an array is an example of a one-to-many selection, and reading from an array is an example of a many-to-one selection. One simple way to profile index statistic is to observe the sequence of values on an index variable from the state-change stream using the library procedures introduced in Section 3.5.3: `chpsim-state-trace-focus-reference`, and `chpsim-state-trace-single-reference-values` (listed in Appendix F.2.2).

A histogram of index values with sufficiently skewed statistics (mathematically, low entropy) can suggest an opportunity to optimize by restructuring an array access.

Every deterministic selection (branch) in CHP has a selection event that fans out to one of several branches, and eventually re-converges at the end of selection. If the performance of the branch event depends on the number of branches, then it is worth studying whether or not the distribution of branches taken is balanced or skewed. A distribution heavily favoring one branch may be grounds for restructuring all other cases to a second-level branch. One such tool that can profile branches is TAST, from TIMA, which specifically looks for restructuring opportunities [60]. TAST, however, does not provide a general framework for constructing arbitrary trace analyses; users are limited to analysis routines provided by the tool authors.

The other common selection structures found in asynchronous circuits are *splits* and *merges*, which route channels one-to-many and many-to-one respectively (Appendix B.4). These structures are often found on multi-level buses and datapaths in existing processors [30, 43]. Splits and merges are somewhat unique to asynchronous circuits; it is important to distinguish them from multiplexers (muxes) and demultiplexers (demuxes). Conventional muxes have no notion of channel handshaking, they passively forward a signal from many-to-one in combinational logic, while non-selected inputs are ignored. Without handshaking, outputs are simply wire-copied to multiple destinations, where they can either be used or ignored. An asynchronous merge, however, enables an output channel to receive a token from one of several input channels, while *blocking* communication on all other inputs channels. An asynchronous split forwards a token from an input channel to one of many output channels, while stopping communication on non-selected output channels. Each iteration of a split or merge process also consumes the control token that selects the channel. To summarize, splits and merges operate on

asynchronous channels and tokens, whereas muxes and demuxes operate on values.

The performance of single-stage splits and merges is determined by the number of input or output channels. Larger structures have greater internal capacitance and consequently can achieve slower peak throughput. Splits and merges can be decomposed into multi-level structures to improve the overall achievable throughput, but at the cost of forward latency. Restructuring must be justified by the statistics of the selection and the criticality of each case. Cases that are more throughput-critical favor finer decompositions, but cases that are more latency-critical should be closer to the root of the structure. Our trace analysis framework gives the ability to study criticality of split and merge structures in sufficient detail to justify restructuring them as optimizations.

4.6 Replication vs. Coalescing

The last set of transformations we examine emphasizes the limitless possibilities of parallel program transformations available to asynchronous circuit designers. It is the overwhelming choices of equivalent high-level programs that motivate an analysis framework to tame the otherwise intractable exploration of design spaces. The role of trace analysis is to prune the space of local transformations to explore. This section focuses on opportunities to replicate or share processes in parallel programs.

Perhaps the greatest motivation for sharing structures is area reduction, which can be significant if the program consists of repetitive and under-utilized structures. Area and circuitry reduction directly leads to reduced static power dissipation, which is accounting for a larger fraction of integrated circuits' power budgets as transistor feature sizes continue to shrink. The question a designer asks is: when is it legal and beneficial to *share* a process, where one process performs the

duty of multiple identical processes? In this context, 'process' refers loosely to any computation that can be modularly factored out (and can be re-used), such as expressions. For example, since integer multiplication and division are costly in area, designers often seek to share a few number of units across an entire design. Integer arithmetic and logical operations are common inside microprocessors, however, since they are small and frequently used, many instances can be found in typical designs.

The software abstraction of executing code gives no cost to calling the same (side-effect free) procedure from multiple (even concurrent) call sites; every procedure call executes in its own frame, there is no conflict in re-using the same procedure code. Without the abstraction of an execution frame, hardware design presents interesting design tradeoffs in process sharing. Software compilers have a similar challenge in procedure inlining. Inlining can bring performance improvements by reducing call-return overhead, but at the cost of code bloat from replication. Excessive inlining can reduce instruction locality and cache performance. Decisions to inline code can be aided by profiling the importance of each call site at run time. Likewise, profiling the execution of concurrent hardware can lead to better decisions to replicate or share processes.

4.6.1 Temporal Activity Analysis

It can be very difficult to statically analyze a massively parallel program to determine exactly when a process *may* be used. A process is said to be *in use* when it is *not* idly waiting for channel inputs[2]. In terms of event graphs (and simulated execution, Section 3.2.1), a process is not in use when its only active events are blocked-waits on channel receives; at all other times, a process is said to be in use.

[2]We restrict our attention to only processes that communicate strictly over channels, and not through shared variables.

Given this definition, one can construct *temporal analyses* that operate on the simulation trace files. The general outline of a temporal analysis is:

1. define initial state object S
2. for every event e in trace (forward in time)
3. update state S with procedure P on event e

The visiting state object S can keep track of any information from the event stream, including whole subsets of the stream. A procedure for temporal analysis will record times at which the user-defined state changes depending on the events seen. For example, the following procedure outline targets all processes of a certain type, and determines when each instance is 'in use':

1. given trace event e and state S
2. let p be the index of the process to which e belongs
3. if type $T(p)$ is the type of interest
4. if e changes the tracked in-use state S of process p
5. append to history of state change new state and timestamp $t(e)$
6. end if
7. end if

The utilization profile accumulated in S contains a series of times at which each process' state changed — in this context, state corresponds to whether or not a process is 'in use.' The utilization profile per instance informs a designer (or synthesizer) of possibilities for sharing one process among multiple locations, also known as time-multiplexing. A set of instances whose utilization profiles do not overlap are potentially good candidates for sharing or coalescing[3]. Conversely, a

[3]The utilization profile however lacks information about the physical locality of the examined processes, which can account for the communication cost of sharing. Post-placement information would also be useful in filtering candidates for sharing.

process that is constantly in use and on the critical path may be a good candidate for replicating to improve throughput.

Temporal analysis is just a general concept with a wide variety of applications. For studying pipeline dynamics, temporal analysis can be used to collect statistics on the average occupancy of a FIFO, which can help slack matching or FIFO restructuring. Activity analysis can be used to construct energy profiles and evaluate local power dissipation over a sliding window of time. In this section, we focus on activity profiling as a means of determining where it may be beneficial to replicate or share processes.

4.6.2 Coalescing Transformation

The search for process-sharing opportunities can start by statically examining all expressions in the entire CHP parallel program. All instances of the same expression operators can be identified using static analysis. Larger compound expressions can be also be found searching beyond single operators[4].

Initial specifications for asynchronous circuits are often un-decomposed and express little concurrency. Consequently, sequential specifications often contain identical (but independent) instances of the same computation in one large outer loop. Consider the following examples with multiple multiply-and-accumulate (MAC) expressions:

```
define MACSEQ ≡
*[...;
  x := a · b + c;
  ...;
  y := d · e + f
 ]
```

[4]This differs from common subexpression detection and elimination in software compilers, where the leaves (literals) of common expressions must match — in this context, we are only interested in common expression *structures*.

```
define MACCOND ≡
*[...;
   [g → ...; x := a · b + c; ...
   []else → ...; y := d · e + f; ...
   ]
 ]

define MACPARA ≡
*[...;
   {x := a · b + c; ...}, {y := d · e + f; ...};
   ...
 ]
```

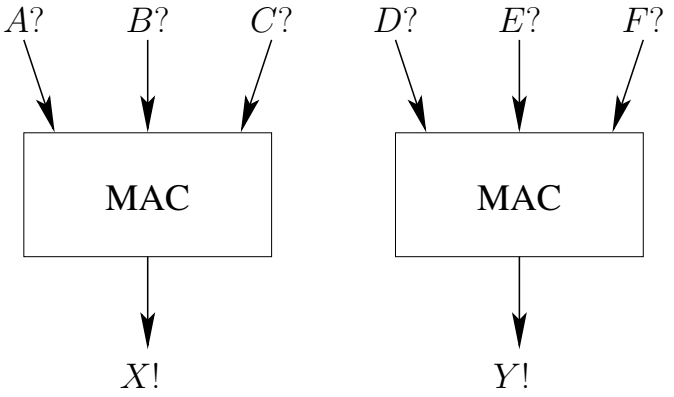


Figure 4.6: Independent, replicated function units can operate concurrently.

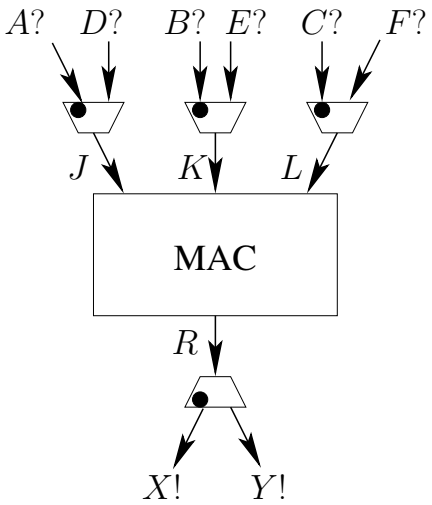


Figure 4.7: Single function unit shared in alternation

The *MACSEQ* process uses the MAC operation twice in sequence per loop iteration, the *MACCOND* process uses the MAC operation once per iteration in an exclusively guarded clause, and the *MACPARA* contains two explicitly concurrent operations. In all of the above processes, there is opportunity to re-use a single MAC unit. If we *project* (cf. projection [41]) variable definitions into explicit channel communications, *MACSEQ* would be rewritten like:

```
define MACSEQ.a ≡
*[A?(a), B?(b), C?(c);
   x := a · b + c;
   X!(x);
```

```
  ...;
  D?(d), E?(e), F?(f);
  y := d · e + f;
  Y!(y)
]
```

Both instance of the MAC operation operate on independent variables, i.e., there are no flow dependencies between them, so one can "factor out" a common process definition, *MAC*. Figure 4.6 shows two instances of *MAC*s operating concurrently and independently. After factoring out two independent MAC operations, *MACSEQ* looks like:

```
define MACSEQ.b ≡
*[A!(a), B!(b), C!(c);
  X?(x);
  ...;
  D!(d), E!(e), F!(f);
  Y?(y)
 ]

define MAC(A?, B?, C?, X!) ≡
*[A?(a), B?(b), C?(c);
  x := a · b + c;
  X!(x);
 ]

MAC P, Q;     // instance declaration
P(A, B, C, X); // port connection
Q(D, E, F, Y);
```

Recall from the original sequential program for *MACSEQ*, that the MAC operations occur in strict alternation. Although decompositions of the program may gradually add concurrency (Section 4.1), it is also correct to enforce sequential alternation in the original specification decompositions. Sequentializing transformations that reduce concurrency are legal as long as the interface semantics are preserved. If we maintain strict alternation, then we guarantee that the two instances of MAC will never be in use at the same time. Perfectly interleaving

utilization is an ideal situation that can warrant sharing of two or more processes. One way to share a process whose use is strictly alternating is to wrap split- and merge-alternators around the interface channels, shown in Figure 4.7. Alternators are also known as round-robin structures. A generic merge-alternator is listed as Program B.12, and a split-alternator is listed as Program B.11. Coordinated sets of alternators can redirect inputs and outputs to a process to *time-multiplex* its use through replicated interfaces. The alternating implementation of Figure 4.7 can be defined as follows:

```
define ALTMAC(A?, B?, C?, D?, E?, F?, X!, Y!) ≡
MAC  P(J, K, L, R);
*[{A?(a); J!(a)}, {B?(b); K!(b)}, {C?(c), L!(c)};
   R?(x); X!(x);
   {D?(d); J!(d)}, {E?(e); K!(e)}, {F?(f), L!(f)};
   R?(y); Y!(y);
 ]
```

or in terms of split- and merge-alternators (Figure 4.7):

```
define ALTMAC(A?, B?, C?, D?, E?, F?, X!, Y!) ≡
MAC  P(J, K, L, R);
ALTMERGE  _(A, D, J);
ALTMERGE  _(B, E, K);
ALTMERGE  _(C, F, L);
ALTSPLIT  _(X, Y, R);
```

The area saved is roughly the area of one *MAC* process, if the area of the alternators is small in comparison.

In the *MACCOND* process, once instances of the MAC operation have been identified, the *MAC* process can be factored out of exclusive branches:

```
*[...;
   [g → ...; A!(a), B!(b), C!(c); X?(x); ...
   []else → ...; A!(d), B!(e), C!(f); X?(y); ...
   ]
 ]
MAC  _(A, B, C, X);
```

The occurrences of the same channel communications in multiple branches are

implicitly controlled splits and merges, which are simpler to implement than alternators because they contain no state between iterations. In this case, the control for the splits and merges is the boolean guard g.

Note that had we re-written *MACSEQ* using the same channels:

```
define MACSEQ.c ≡
*[A!(a), B!(b), C!(c);
  X?(x);
  ...;
  A!(d), B!(e), C!(f);
  X?(y)
 ]

define MAC(A?, B?, C?, X!) ≡ ...
MAC P;
P(A, B, C, X);
```

there would be an implicit alternators on the channels interfacing to *MAC*. The *MACSEQ* and *MACSEQ.c* programs can be written (or internally represented) as definition *MACSEQ.b* to articulate def-use chains of unrelated variables for dataflow analyses.

We can also take a program with explicit concurrence such as *MACPARA* and sequentialize use of common resources because $S; T$ is a legal execution of S, T. Even in cases where dynamic activity profiling shows frequent concurrent use of the same type of resource, designers have the option to trade off performance for area reduction by sharing resources among multiple uses. All of the aforementioned equivalent definitions of *MACSEQ* are valid implementations of *MACPARA*; sharing a resource (in this case, with alternation around a functional unit) can save considerable area from large resources. Without quantitative measurements from trace analysis of different program refinements, it is difficult to evaluate the area-performance tradeoffs with resource sharing or replication.

Replication. A parallel program may also be written initially with explicit sharing, so searching for opportunities to replicate overloaded units to improve

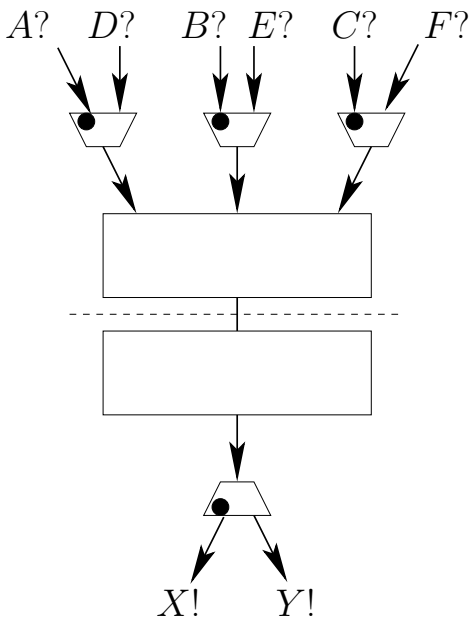


Figure 4.8: Pipelined function unit, shared in alternation

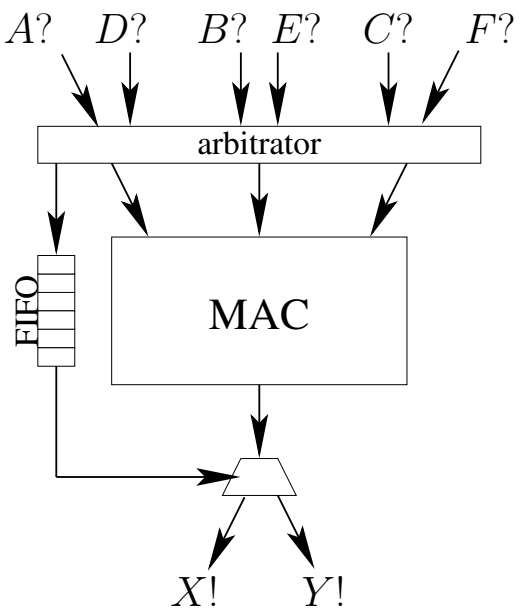


Figure 4.9: Single function unit shared by arbitration

performance is also important. Since many programs are input-dependent, overloaded units often cannot be recognized until their execution is traced and profiled. The immediate benefit of replication is increase in achievable throughput; an application whose performance is bounded by operation bandwidth (e.g. MAC) can alleviate its bottleneck by using more units in parallel. Processes that have been identified with high utilization and frequently occur on the critical path are likely to benefit in performance after replication; the additional area cost can be justified by speedup.

Combining transformations. Sharing one process among multiple uses (in the interest of reducing area) can possibly reduce the throughput of the whole program. After sharing a structure, profiling may reveal that a new critical path limits the repetition rate of the structure. One may be able to recover some of the lost performance by pipelining the shared computation structure, as discussed in Section 4.2 (Figure 4.8). Pipelining a shared structure will allow it to support mul-

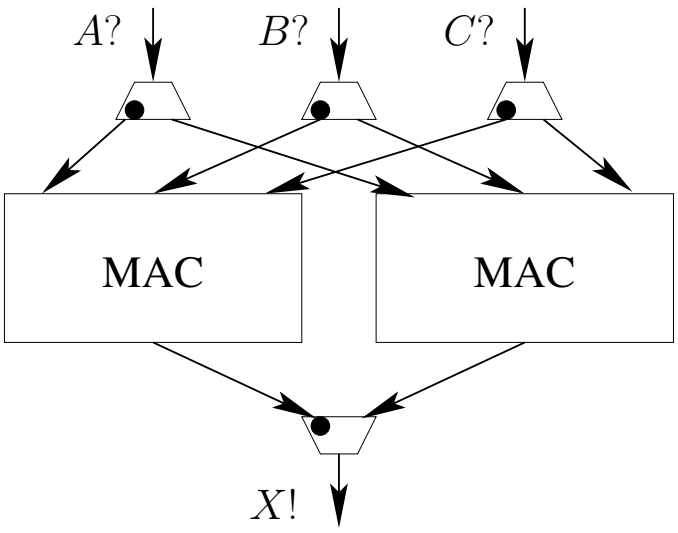


Figure 4.10: Internally replicated function units can be accessed through alternators with a single-unit interface.

tiple concurrent operations and shorten internal critical paths, thereby improving throughput. Pipelining does incur an overhead cost in communication circuits, but the overhead is justifiable if it is insignificant compared to the area of the original unpipelined structure.

Another method for replicating processes uses alternators to dispatch to *locally* replicated processes in a round-robin order, as shown in Figure 4.10. This transformation is legal because results are still produced in the same order that inputs are received, i.e., FIFO order is maintained. The result is a single interface to a process, whose internal replication effectively increases throughput and concurrency. Local replication is useful when it is impractical to pipeline a particular process. Simulation and trace analysis can be used to weigh the area cost of replication against the speedup gained.

4.6.3 Nondeterministic Dispatching

In practice, one will find much larger and more complex programs than our little multiply-and-accumulate operator examples. Static analysis of parallel programs can quickly become unwieldy because the set of distributed states that *can* be reached from any given state grows exponentially with the number of concurrent actions per fork. (Recall that parallel programming requires no timing assumptions, and thus, cannot rely on timing assumptions to partially order concurrent actions. Conservative families of asynchronous circuits, such as QDI, rely on only modest timing assumption which only negligibly reduce the space of reachable states.) In many cases, static analyses will not be able to infer any ordering relations between uses of identical processes. To share a process among different uses without any knowledge of sequencing, a designer can resort to *arbitration* or dynamic dispatching. Arbitrated dispatching, which is akin to dynamic task allocation in parallel software, can be useful when the computation of interest has greatly varying or input-dependent timing characteristics [16]. Without arbitration, programs are restricted to operating in strict order, i.e., values on channels will be ordered deterministically. The next example demonstrates how arbitrated resource sharing can be useful in deterministic parallel programs without timing variations. Consider the following program that conditionally computes up to two MAC results per iteration:

```
define  MULTIMAC ≡
*[...;
    {[g1 → x := a · b + c; ... []else → skip],
      [g2 → y := d · e + f; ... []else → skip]};
    ...
  ]
```

Since use of the MAC operations is not ordered or exclusive, synthesizing the program may require up to two MAC units. However, the program can be rewritten

using only one *MAC* unit with arbitrated sharing. In cases such as this, the identical computations can be coalesced into a single functional unit, shared by arbitration, as shown in Figure 4.9. The arbitration process forwards one set operands to the shared functional unit when all inputs from a set are ready. The result of arbitration is also forwarded to output merge after the functional unit, which produces results in the same (FIFO) order from arbitration. The resulting process behaves like multiple functional units, but uses only one overloaded unit. An important characteristic to note is that the nondeterminism from arbitrated sharing is *localized* to one process alone; the effects of local nondeterminism do not disturb the interaction with the rest of the parallel program. Non-invasiveness is generally preferred when exploring local program transformations as it requires minimal rewriting.

In *MULTIMAC*, during iterations where two MAC operations are required, they will be sequentialized by arbitration, which still results in legal executions of the concurrent specification, at the cost of some performance. When only one MAC operation is needed, there is no contention, and the design pays only the performance overhead of the wrapper arbitration and merge. This transformation can be appealing if the number of functional units shared meet demands most of the time without sequentialization penalty, i.e., the sequentialization scenario rarely occurs on the critical path. If the shared function unit can be pipelined, then the overloaded unit can support multiple concurrent computations (represented in Figure 4.8) and gain back some performance. Next, we show how resource sharing can be generalized to pooling using arbitration.

Resource pooling and partitioning. By combining local arbitrated dispatching with resource replication, one can create a pool of M identical resources shared among N users. One way to organize a pool is to replace the shared *MAC*

process in Figure 4.9 with the alternator structure in Figure 4.10. Pooling is an appealing option when the capacity of a single shared unit does not meet the average demand from multiple users. However, the channel between input arbitration and alternator dispatch can become a new bottleneck; every action must pass through a single channel (or set thereof), which scales poorly with size. A typical solution to congesting channels is to *partition* the users and resources (and channels) into K disjoint sets, effectively distributing contention among K sequentialization points. Different partitioning schemes exhibit a tradeoff between performance and resource contention; a partitioned set of resources is less capable of load-balancing than a unified set of resources. Activity profiling can justify the cost of pooling resources, and help find reasonable static partitionings. In an ideal situation, profiling may reveal a partitioning that suitably balances load among all partitions. This class of problems involving design space exploration of resources sizes and partition topologies is ubiquitous in computer architecture.

The choice to use shared arbitration can only be justified by studying the runtime dynamics from simulation. Again, a temporal activity analysis of a program can provide insight about the extent of local resource contention one might expect from sharing transformations. The results from temporal analyses can be used to determine which occurrences of the same expression can share the same function unit with minimal contention. Looking for minimal activity-time overlap is only a simple heuristic for clustering operations into a limited number of functional units. However, computation activity may overlap in situations where the results are not on the critical path, that is, the result produced has sufficient slack time to delay before it becomes critical. By combining critical path slack-time analysis (Section 3.6.4) with activity overlap analysis, one can better estimate the performance impact of sharing instances of function units.

4.6.4 Arbitration with Reordering

.

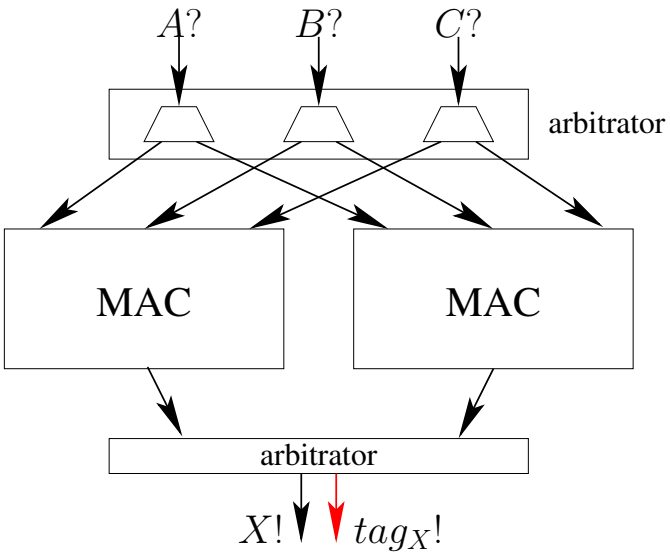


Figure 4.11: Arbitration can be used to dispatch operands to first-available processes and to reorder results from replicated units.

Previous examples have worked under the implicit assumption that all *MAC* operations take equal time, however, that is not the case for general computations. Variable-latency operations introduce opportunities where arbitration may speed up programs over those without arbitration. An example of a variable latency computation is an iterative multiplication (or division), where the number of iterations depends on the multiplicand (or divisor). Without re-ordering, results are computed first-in-first-out (FIFO), even when work is distributed in a round-robin order by alternators as in Figure 4.10; results that are quickly computed must still wait behind longer operations at the result merge. If earlier results were allowed to *overtake* the later results from previous iterations, some performance could be gained. Figure 4.11 shows how one might wrap around replicated processes to allow results to be dynamically reordered. Operands are passed to the process through one set of interface channels and dispatched to the 'first-available' pro-

cess, by arbitrating the channel acknowledgments. Output arbitration will pass results through first-come-first-serve (FCFS), not necessarily in the same order that computation was dispatched. This transformation is not entirely transparent; reordering requires that output arbitration to track and communicate its selections (such as channel tag_X in Figure 4.11), so that results can be correctly matched up with the corresponding sequence of inputs. The surrounding processes involved in this replication scheme need to be transformed to sort out the results based on channel tag_X. This arbitrated replication scheme allows one of the internal units to take a longer time without blocking fast operations on other free internal units. The cost of this transformation (area and arbitration overhead) needs to be justified with trace statistics:

- Does the computation exhibit variable delay-to-result?
- Does the critical path show potential for benefit from reordering? In other words, are there large slack times available? (Section 3.6.3)

4.7 Application to Synthesis Optimization

Design choices for resource replication and sharing fall in the domain of general synthesis problems. What a designer writes in high-level CHP need not be interpreted literally; synthesizers are free to implement many circuit details and structures. Automated synthesis can be optimized more effectively with statistics from trace profiling, and even preliminary placements and routings (to exploit physical locality).

One of the original weaknesses of conventional syntax-directed and data-driven translations was the inability to re-use instances of the same computation [7, 77]. With a detailed activity profile and analysis, one could potentially annotate and direct the synthesis to coalesce re-usable computations that consume much area. Data-driven and syntax-directed translation often results in circuits that are *overly*

decomposed, resulting in excessive pipelining which may run into area or resource constraints. One approach to post-translation clustering (re-composing) processes optimizes clustering along critical path [78]. Their approach is concerned with fusing computations into a single PCHB stage, which is akin to de-pipelining. Their decisions to combine or pipeline computations are based on the characteristics of the resulting PCHB circuits, such as the expected performance of the precharge logic relative to a target cycle time.

The span of program transformations that can be applied to concurrent programs is innumerable; it is infeasible to consider the entire set of equivalent programs under *all* possible transformations. Many local transformations are orthogonal and can be applied independently of one another, which results in an exponential number of equivalent versions to consider.

Profile analysis can help reduce the set of transformations to consider, prioritize candidate transformations, and guide transformations in the correct direction. Our trace query framework can be used to assemble arbitrarily complex and extensible analyses, which can then be used interactively or automatically for program rewriting iterations. Interactivity is essential for a user to be able to dynamically construct a series of queries during diagnostic sessions.

This section presented a few classes of transformations, and contexts in which they are not obviously beneficial without run-time profile information. In each scenario, we outlined the analyses that one would construct from trace query primitives, to inform a designer (or compiler) where to focus optimization efforts and in which direction transformations should be applied.

CHAPTER 5

APPLICATIONS: CASE STUDIES

However smart a robot or computer may be, it can only do exactly what you tell it to do and then stop. To keep thinking, it has to want to. It has to be motivated. You can't think if you can't feel. So the ship's intelligence had to be imbued with emotions, with personality. And its name was Titania.

Terry Jones, "Douglas Adams's Starship Titanic"

We have shown how trace profiling of high-level parallel programs (targeting asynchronous circuits) can be crucial to identifying and selecting appropriate program transformations for optimization or trading off between metrics. To demonstrate the utility of our analysis framework, we analyze a series of small design-space exploration problems. Each case exhibits different design tradeoffs and optimization problems that are encountered in practice.

5.1 Fibonacci Generator

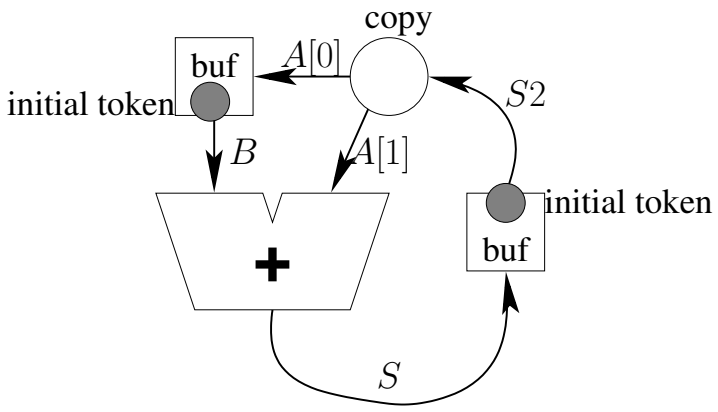


Figure 5.1: Schematic of a decomposed Fibonacci sequence generator

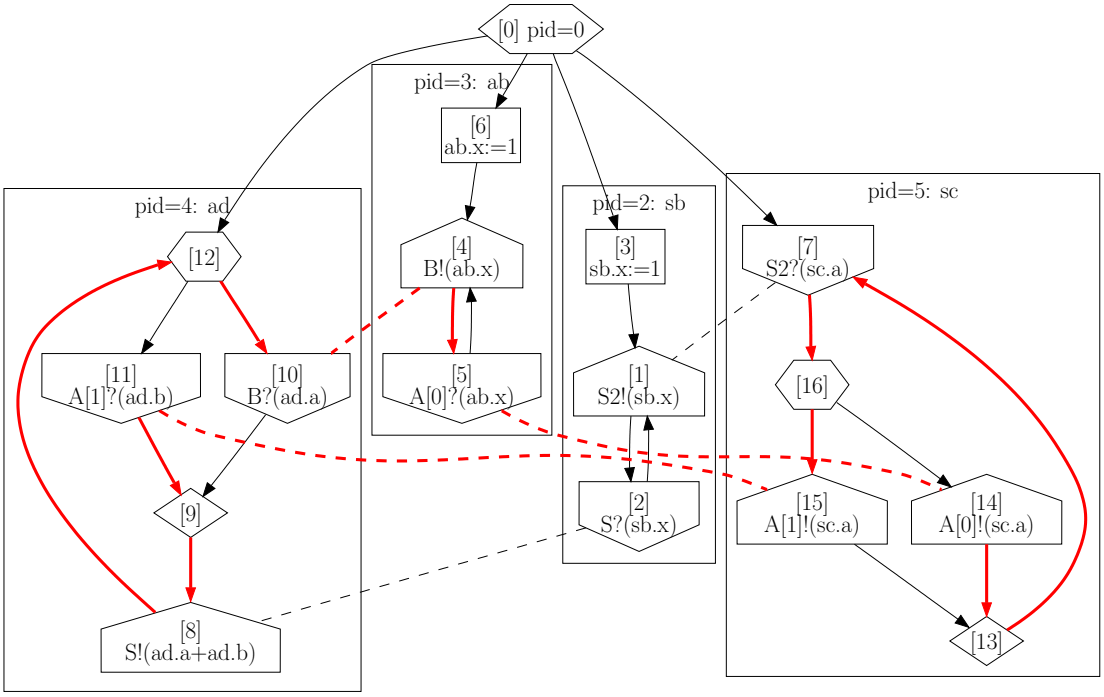


Figure 5.2: CHP event graph of initially decomposed Fibonacci sequence generator. Bold-red edges mark the critical path from Table 5.1.

The first program is a simple Fibonacci sequence generator, shown in Figure 5.1. A sequential specification in CHP is:

$a; = 1, b := 1;$
$*[c := a + b;$
$\quad a := b;$
$\quad b := c$
$\quad]$

Even though this process is closed (no ports), one could easily copy a variable to an output channel on every iteration to obtain a sequence of values. We omit such a channel because, for simulation purposes, it would simply be directed to a token sink and discarded. In this program, variables a and b have loop-carried dependencies, whereas c can be a local variable because it is dead (in the dataflow sense) past the end of the loop. A finely decomposed version of the loop can be written:

$*[A[1]?(a), B?(b); S!(a + b)]$

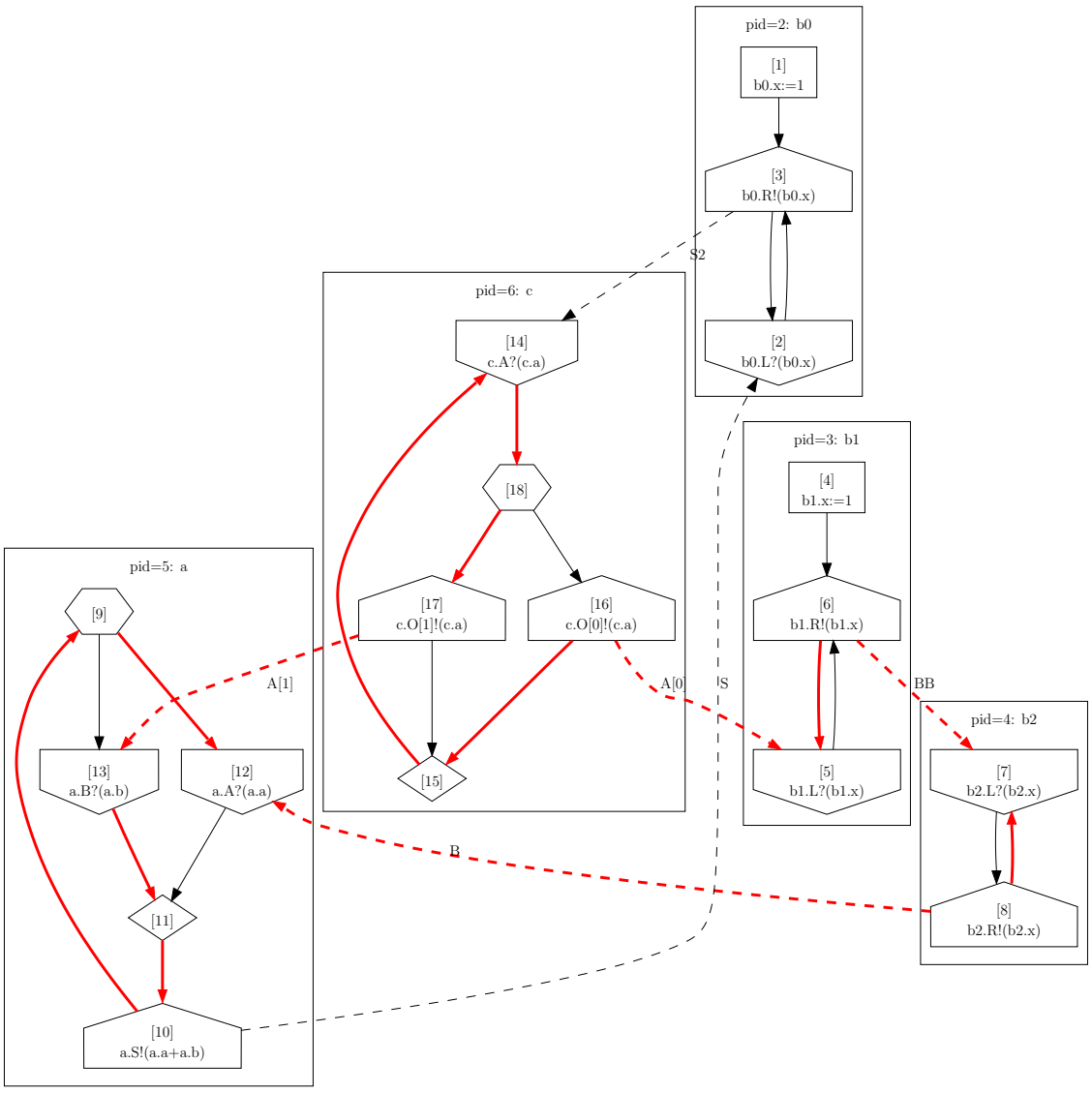


Figure 5.3: Event graph of a partially slack-matched Fibonacci sequence generator. Bold-red edges mark the critical path from Table 5.2.

$$
\begin{array}{ll}
\| \; s := 1; *[S2!(s); S?(s)] & // \;\; \textit{initial token buffer} \\
\| \; *[S2?(o); A[0]!(o), A[1]!(o)] & // \;\; \textit{copy} \\
\| \; a := 1; *[B!(x); A[0]?(x)] & // \;\; \textit{initial token buffer}
\end{array}
$$

The expanded CHP event graph of the initial decomposed program is shown in Figure 5.2. The decomposed processes assume the following delays in simulation:

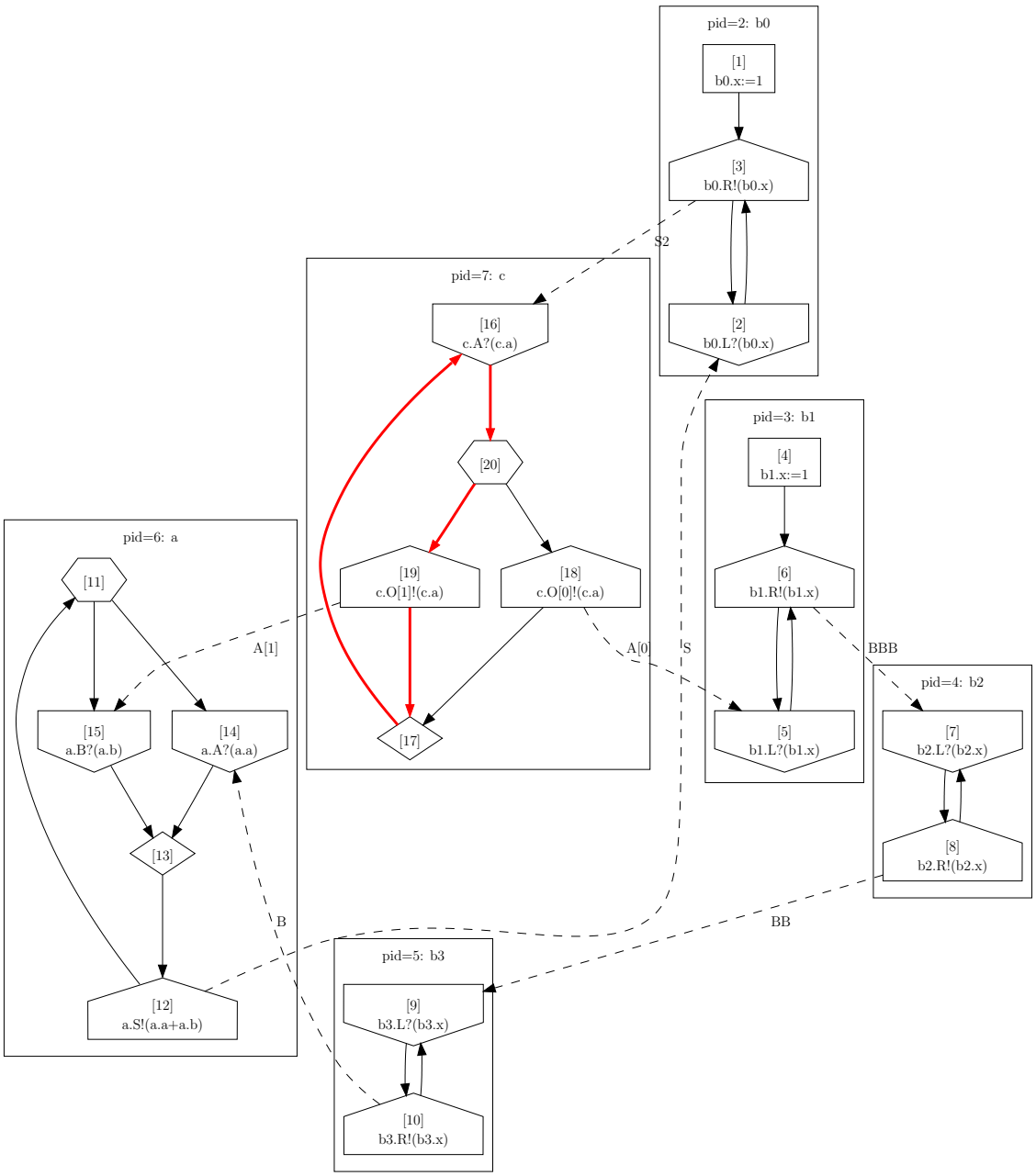


Figure 5.4: Event graph of a fully slack-matched Fibonacci sequence generator. Bold-red edges mark the critical path from Table 5.3.

process	forward latency	backward latency
adder	3	5
copy	2	6
buffer	2	5

Forward latency is the time delay from the moment that the send action in each process is "ready to execute" to the time that the event actually occurs. Backward latency is the analogous delay for receive actions. A lower bound for the cycle time of the composition of all processes is the maximum target cycle time of any process, which is the adder, with 8 (3+5) time units, and the copy, with 8 (2+6) time units.

Even without intimate knowledge of a particular design, an engineer who is asked to optimize this design for performance can start with a critical path analysis on a simulation trace (Section 3.6, Program F.4). Table 5.1 shows an excerpt of the critical path from simulating the initial decomposed version of the Fibonacci loop. The table is organized as described in Section 4.2.2.

This initial version of our decomposed Fibonacci generator has a cycle time of 11 time units (from observing the repetition time of any single event); this design fails to achieve the minimum cycle time of 8 units. We analyze the critical path for slack matching problems in the same manner as in Section 4.2.2. To emphasize the critical events that cross process boundaries through channel communications, we have paired critical send-receive events together in Table 5.1. The send-receive event pairs in Figure 5.2 (with named channels) are: S (8,2), A[0] (14,5), A[1] (15,11), $S2$ (1,7), B (4,10). We query whether these channels are sender-critical or receiver-critical by examining paired sends and receives on the critical path (Program F.12):

```
; 'crit' is the critical path stream
hacchpsimguile> (channel-send-receive-criticality crit "A[0]")
(0 . 7)
hacchpsimguile> (channel-send-receive-criticality crit "A[1]")
(7 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "B")
(0 . 7)
hacchpsimguile> (channel-send-receive-criticality crit "S")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "S2")
(1 . 0)
```

Table 5.1: Critical path through a minimum-slack Fibonacci loop (Figure 5.2)

index	time	event	crit.
92	76.0	16	90
90	75.0	7	87
87	70.0	13	85
85	70.0	14	84
84	70.0	5	77
77	65.0	4	76
76	65.0	10	73
73	60.0	12	70
70	59.0	8	67
67	57.0	9	66
66	57.0	11	65
65	57.0	15	64
64	55.0	16	62
62	54.0	7	59
59	49.0	13	57
57	49.0	14	56
56	49.0	5	49
49	44.0	4	48
48	44.0	10	45
45	39.0	12	42
42	38.0	8	39
39	36.0	9	38
38	36.0	11	37
37	36.0	15	36
36	34.0	16	34
.....			

Table 5.2: Critical path through a partially slack-matched Fibonacci loop (Figure 5.3)

index	time	event	crit.
159	95.0	16	158
158	95.0	5	149
149	90.0	6	148
148	90.0	7	138
138	85.0	8	137
137	85.0	12	133
133	80.0	9	129
129	79.0	10	128
128	77.0	11	126
126	77.0	13	125
125	77.0	17	122
122	75.0	18	118
118	74.0	14	114
114	69.0	15	111
111	69.0	16	110
110	69.0	5	101
101	64.0	6	100
100	64.0	7	90
90	59.0	8	89
89	59.0	12	85
85	54.0	9	81
81	53.0	10	80
80	51.0	11	78
78	51.0	13	77
77	51.0	17	74
74	49.0	18	70
70	48.0	14	66
66	43.0	15	63
63	43.0	16	62
.....			

Among these channel events, the only send-receive actions that appear *paired* on the critical path are A[0] (14,5), A[1] (15,11), and B (4,10). Among these pairs, A[0] and B are always receiver-critical, and A[1] is always sender-critical. (S and

$S2$ are not latency critical.) From this, we deduce that the path through channel B has sufficient slack (buffering), and the latency through A[1] cannot be reduced because there are no buffers to remove on that path. Since the path through A[0] and B is always blocked waiting for the receiver, increasing buffering is likely to improve overall performance. The result of the analysis directs the designer to try adding buffering on the path through channels A[0] and B.

Table 5.3: Critical path through a fully slack-matched Fibonacci loop (Figure 5.4)

index	time	event	crit.
98	54.0	16	89
89	49.0	17	87
87	49.0	19	82
82	47.0	20	80
80	46.0	16	71
71	41.0	17	69
69	41.0	19	64
64	39.0	20	62
62	38.0	16	53
53	33.0	17	51
51	33.0	19	46
46	31.0	20	44
44	30.0	16	35
35	25.0	17	33
33	25.0	19	28
28	23.0	20	26
26	22.0	16	18
.....			

In our second revision of the Fibonacci generator (Figure 5.3), we add one more buffer on channel B (or equivalently, A[0]), which results in a cycle time of 9 time units, still shy of peak performance. The new critical path is shown in Table 5.2. Critical path analysis on channel events indicates that the path through A[0], BB, and B is still receiver-critical and is limiting the performance, like the initial design. The analysis suggests adding more slack on the receiver-critical path.

```
; 'crit' is the critical path stream
hacchpsimguile> (channel-send-receive-criticality crit "A[0]")
(0 . 4)
hacchpsimguile> (channel-send-receive-criticality crit "A[1]")
(3 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "B")
(0 . 3)
hacchpsimguile> (channel-send-receive-criticality crit "BB")
(0 . 3)
hacchpsimguile> (channel-send-receive-criticality crit "S")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "S2")
(1 . 0)
```

The third revision adds yet one more buffer on channel B (Figure 5.4) and achieves the minimum cycle time of 8 units. The new critical path is shown in Table 5.3,

```
; 'crit' is the critical path stream
hacchpsimguile> (channel-send-receive-criticality crit "A[0]")
(1 . 1)
hacchpsimguile> (channel-send-receive-criticality crit "A[1]")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "B")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "BB")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "BBB")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "S")
(0 . 0)
hacchpsimguile> (channel-send-receive-criticality crit "S2")
(0 . 0)
```

The set of recurring events in the new critical path (16,17,19,20) no longer crosses process boundaries and corresponds to only events in the adder process, whose cycle time is 8. The channel criticality analysis finds that there are no repeated occurrences of send-receive event-pairs on the critical path; the design's performance is limited by the throughput of a single process, the adder.

```
; critical path contains mostly events 16, 17, 19, 20
hacchpsimguile> (map hac:chpsim-event-process-id
   '(16 17 19 20))
(6 6 6 6)
hacchpsimguile> (hac:parse-reference "fibber.a")
(process . 6)
; this is the adder process
```

This final design is optimally slack-matched; adding or removing of buffers will not improve the performance any further. Table 5.4 summarizes our design revision history with an energy assessment.

Experienced asynchronous VLSI designers probably identified this example as a classic slack matching problem that can be solved with straightforward analysis of steady-state pipeline dynamics. Nevertheless, the analyses developed within our trace analysis framework can lead all designers to the same optimization conclusions from slack matching, but through fundamental critical path analysis. A trace analysis framework can better assess more complicated pipelines that do not conveniently exhibit regular steady state behavior. The interactive analysis interface (Scheme) gives users the ability to examine data and patterns in trace data that might not otherwise be considered in existing generalized analyses.

Table 5.4: Summary of tradeoffs of three designs of Fibonacci loop

rev.	area	energy	cycle time
1	$A_{baseline}$	$E_{baseline}$	11
2	$A_{baseline} + A_{buf}$	$E_{baseline} + E_{buf}$	9
3	$A_{baseline} + 2A_{buf}$	$E_{baseline} + 2E_{buf}$	8

Having compared several versions of the decomposed Fibonacci loop, we can also perform some first-order estimations of the area and energy tradeoffs against performance. The initial decomposition is composed of 1 adder, 1 copy, and 2 initial-token buffers, and the revisions added only 1 and 2 buffers respectively.

These design choices range from an un-optimized design with minimal area and minimal energy per token, to a slack-matched design the greater area and energy per token. We have already found the throughput-optimal solution, but from an energy standpoint, the original design is the most appealing. However, for a mixed metric such as energy-efficiency ($energy \times time_{cycle}^2$ or $E\tau^2$), all three design variations may be good candidates because each revision trades off performance for one of the other metrics. Our analysis framework is a general tool to assist designers in the exploration of design space tradeoffs.

5.2 Bit-serial Routers

The following example emphasizes the importance of simulating and collecting execution traces on designs that are highly input-dependent . The input (or workload) to a system can be characterized by data and timing. The operating characteristics of parallel programs executing under different workloads can heavily influence the cost and benefit of transformations being considered for optimization. Network routers are one class of designs whose design and optimization are heavily dependent on traffic patterns. Our example of a bit-serial model is a significantly simplified circuit that performs bit-serial routing. The following sequential CHP specifies the operation of the bit-serial router:

```
define BITROUTER(L[0..1]?, R[0..1]!) ≡
*[[¬L[0] → L[0]?(lc, dir);
     *[¬lc → L[0]?(lc, ld);  R[dir]!(lc, ld)]
  |¬L[1] → L[1]?(lc, dir);
     *[¬lc → L[1]?(lc, ld);  R[dir]!(lc, ld)]
 ]]
```

To summarize, the router arbitrates between two sources, $L[0]$ and $L[1]$, and beheads the leading bit of each packet; every packet that passes through this router will be shortened by one symbol. The value of the leading bit is the output

destination of the payload of each packet, $R[0]$ or $R[1]$. A packet terminates once it sees a stop-bit, and the cycle repeats. When there are no incoming packets, the process sits idle.

As usual, we decompose this sequential specification to simplify synthesis and increase performance. Since the program is symmetric and shares common action sequences in both cases, one natural way to decompose the bit-router is to perform arbitration in the front process and destination routing in a back process. The resulting program is a composition of a *route-split* and a *route-merge*.

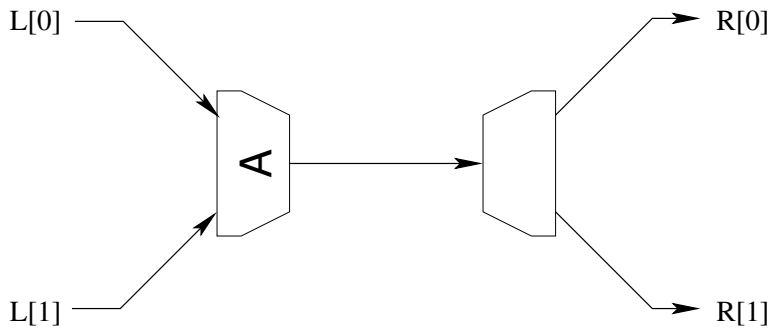


Figure 5.5: Schematic of a decomposed merge-split bit-serial router

```
define ROUTEMERGE(L[0..1]?, X!) ≡
*[lc↓;
  [¬L[0] → *[¬lc → L[0]?(lc, ld); X!(lc, ld)]
  |¬L[1] → *[¬lc → L[1]?(lc, ld); X!(lc, ld)]
  ]
 ]

define ROUTESPLIT(Y?, R[0..1]!) ≡
*[Y?(lc, dir);
  *[¬lc → Y?(lc, ld); R[dir]!(lc, ld)]
 ]

define BITROUTER_decomp(L[0..1]?, R[0..1]!) ≡
ROUTEMERGE M(L, C);
ROUTESPLIT S(C, R);
```

ROUTEMERGE passes arbitrates between incoming packets of bits and forwards stop-bit-delimited packets to a shared output channel. *ROUTESPLIT* consumes the first direction bit of each packet and forwards the rest of the payload to the selected output channel. The composition is shown in Figure 5.5.

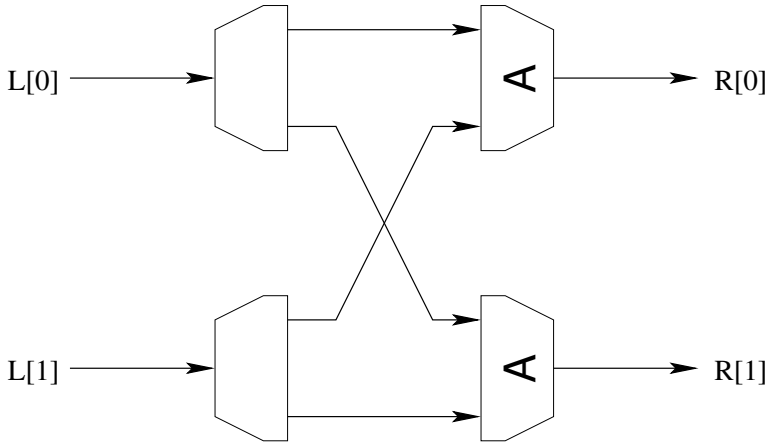


Figure 5.6: Schematic of a decomposed split-merge bit-serial router

Since all packets must pass through the center channel, it can be a possible bottleneck under workloads with significant activity from both input sources. However, for routed packets with different destinations, there is an opportunity to route both packets simultaneously with a different design. The "twin bit-router" in Figure 5.6 interchanges the split and merge phases so that destination routing is done *before* arbitration at the output. The twin design uses twice as many route-splits and route-merges, and adds three more internal channels, and is thus expected to be at least double the area of the single bit-router. The energy per packet in the twin bit-router is comparable: aside from the difference in post-synthesis wire length, the twin bit-router's switching activity per packet is slightly less because the route-split beheads the first symbol earlier and each route-merge handles one fewer symbol per packet.

Before we can conclude which of these two designs is 'better', we compare their execution times under a variety of input workloads in Table 5.5. We connected a

Table 5.5: Execution times of single (Figure 5.5) and twin (Figure 5.6) bit-serial routers under different input workloads

workload	single	twin	% speedup
A	4738	2985	58.727
B	4738	2445	93.7832
C	4757	3137	51.6417
D	4757	3127	52.1266
E	15498	9219	68.1093
F	15555	11969	29.9607
G	15625	15429	1.27034
H	18798	12772	47.1813
I	28352	22438	26.3571

variety of finite workloads to the routers' inputs, connected ideal token sinks to the routers' outputs. Workloads varied by packet length, destination, frequency, and gap time. The twin router design will never perform worse than the single router, and the speedup is limited to 100%. The speedups range from negligible (G) to near-maximum (B) with this set of workloads. We can examine route and resource contention in more detail with some trace analysis procedures from Appendix F.2.7. One expects heavily stressed resources (processes) to appear more frequently on the critical path. We examine critical processes in the twin-bitrouter on workload G:

```
; examining workload G on the twin bitrouter
; crit is the critical path stream
hacchpsimguile> (define proc-histo
   (make-critical-process-histogram crit))
hacchpsimguile> (print-named-critical-process-histogram
   proc-histo)
; process-name: (process-index . count)
BR.RM[0]:  (12 . 98)
BR.RM[1]:  (13 . 90)
BR.RS[0]:  (14 . 29)
BR.RS[1]:  (15 . 1496)
; BR.RM[0..1] are the merges, BR.RS[0..1] are the splits
```

; other processes omitted ...

The analysis finds that even with the twin bitrouter design, for workload G, one split process ($BR.RS$[1]) is by far the most frequent process on the critical path because the routes are not evenly distributed. Upon discovering this bottleneck, a designer who is interested in making workload G run faster can explore transformations on the critical process, such as those discussed in Chapter 4. The framework provides a convenient interface for examining the execution on any benchmark in arbitrary detail.

Clearly, the optimal choice of design depends heavily on the characteristics of the typical workload that the design is expected to encounter. This example demonstrates the importance of having a flexible simulation and trace analysis environment to justify design choices with performance comparisons.

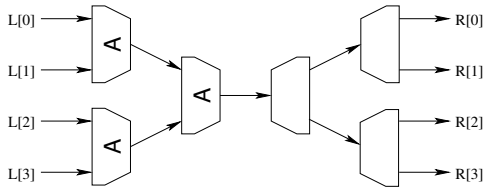


Figure 5.7: A merge-merge-split-split (4,4) bit-router

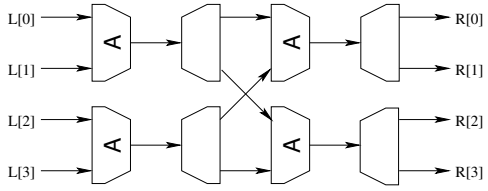


Figure 5.8: A merge-split-merge-split (4,4) bit-router

If we extend our bit-serial router to take 4 inputs and 4 outputs (denoted (4,4)), then we have 6 permutations of route-merge and route-split stages to choose from, shown in Figures 5.7 through 5.12. We name these variations by the sequence of merges and splits encountered as a packet travel from left to right, e.g. 'MMSS'

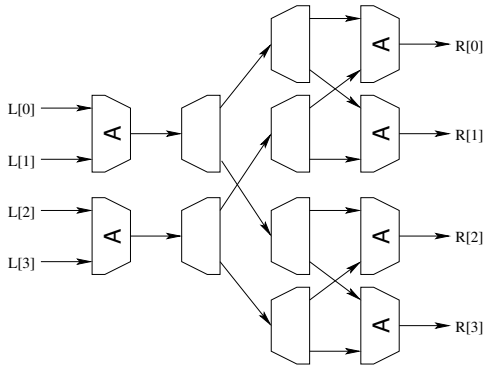


Figure 5.9: A merge-split-split-merge (4,4) bit-router

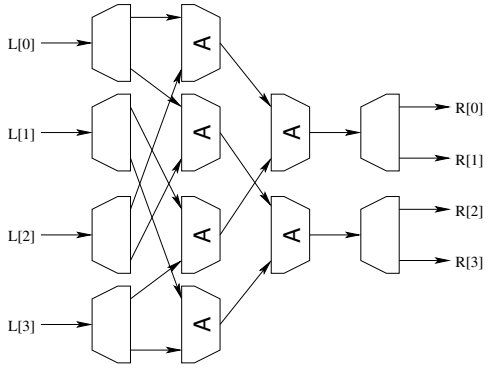


Figure 5.10: A split-merge-merge-split (4,4) bit-router

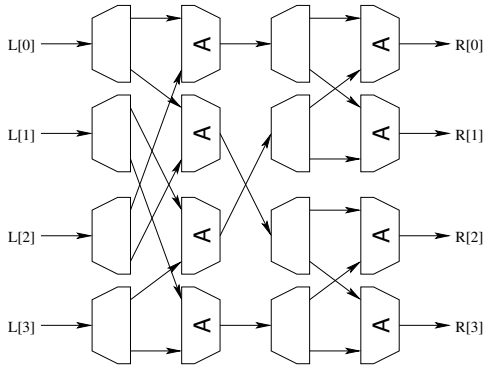


Figure 5.11: A split-merge-split-merge (4,4) bit-router

means merge–merge-split-split, which is our baseline for comparisons. Table 5.6 shows the estimated areas and energy per packet of each design. We also list the number of channels (edges) in this table because routing and interconnect can have a significant contribution to total area. The energy expression consists of two

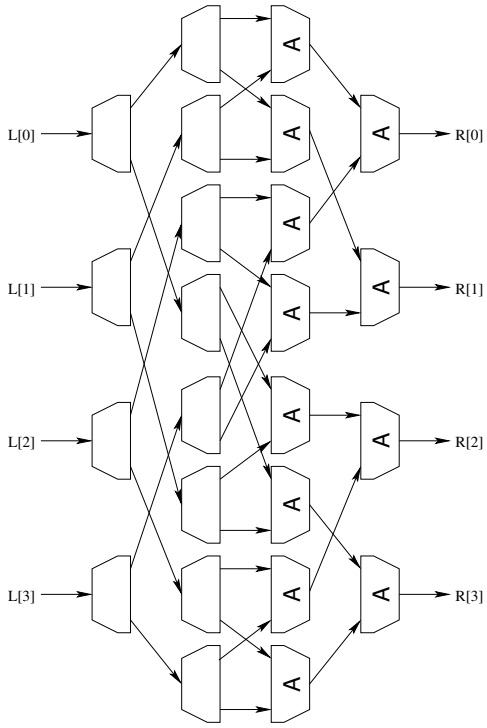


Figure 5.12: A split-split-merge-merge (4,4) bit-router

terms: the energy through route-merges E_m and route-splits E_s. The E_s term is invariant because only route-splits reduce the number of symbols output by one. As packets length increases the relative differences in the E_m terms diminish.

Table 5.6: Area and energy breakdown of various (4,4) bit-routers. K is the total length of a packet (number of symbols), E_s is energy per symbol through a split, E_m is energy per symbol through a merge.

	#splits	#merges	#channels	energy(K)
MMSS	3	3	13	$2KE_m + (2K-1)E_s$
MSMS	4	4	16	$(2K-1)E_m + (2K-1)E_s$
MSSM	6	6	22	$(2K-2)E_m + (2K-1)E_s$
SMMS	6	6	22	$(2K-2)E_m + (2K-1)E_s$
SMSM	8	8	28	$(2K-3)E_m + (2K-1)E_s$
SSMM	12	12	40	$(2K-4)E_m + (2K-1)E_s$

We simulate the (4,4) bit-routers under different workloads and compare their

Table 5.7: Speedups of various implementations of (4,4) bit-serial routers relative to the MMSS baseline, under different input workloads (see also Figure 5.13)

work-load	MMSS +%	MSMS +%	MSSM +%	SMMS +%	SMSM +%	SSMM +%
A	0.0	66.2	78.4	123.1	163.5	235.8
B	0.0	61.3	85.2	120.0	254.1	260.7
C	0.0	61.3	75.9	120.0	254.1	263.8
D	0.0	28.8	58.5	55.6	115.0	163.8
E	0.0	8.4	8.5	8.5	8.6	12.4
F	0.0	63.1	83.2	129.4	182.9	302.2
G	0.0	52.8	79.6	132.0	201.4	255.4
H	0.0	53.2	77.6	98.2	152.3	186.8

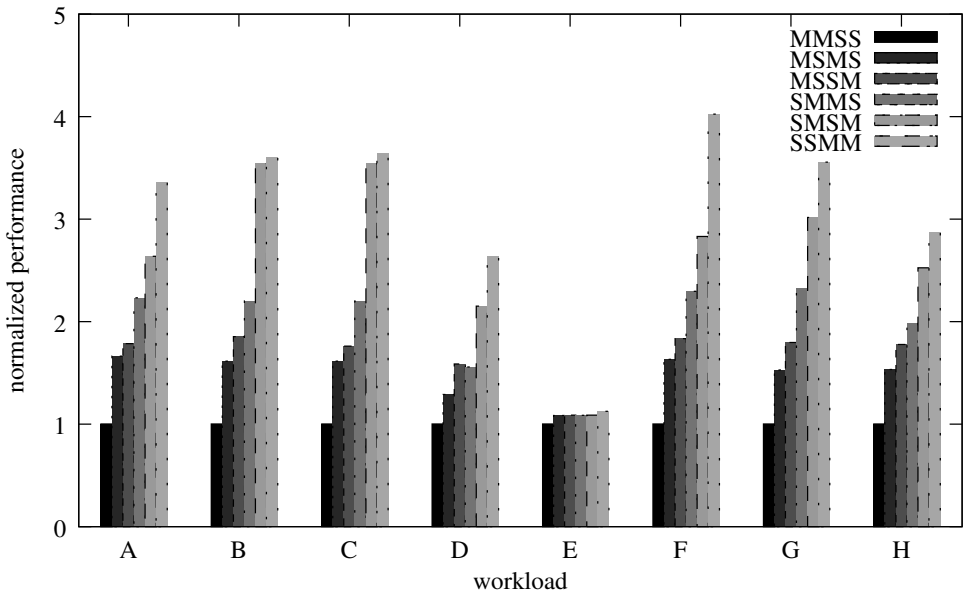


Figure 5.13: Performance of various (4,4) bit-serial routers, normalized to the MMSS baseline (see also Table 5.7)

execution times in Table 5.7 and Figure 5.13. The entries in this table are normalized speedups relative to the baseline design, 'MMSS.' Intuitively, designs with more routing resources will experience less route contention at run-time, however, the benefits are quite staggering under different workloads. Workloads with high destination contention (E) will benefit little from increased routing resources, nor

will workloads with very sparse packets (low load).

An interesting data point is workload F, which shows a super-linear speedup with 'SSMM', with respect to the number of splits and merges (a linear speedup with 4x the area would be 300%). This is explained by the fact that workload F's packet lengths are short enough that reducing packets lengths by one or two (by splits) before the merge stages resulted in reduced contention time and additional measurable speedup.

So far, these initial experiments with different versions of bit-routers are only a preliminary study of router structures. With the trace analysis framework, one can study the execution of each workload in greater detail. For instance, one can continue to look for process bottlenecks with the `make-critical-process-histogram` procedure as before with the (2,2) bit-routers. The bottlenecks should become the focus of subsequent optimization.

Critical path histograms may reveal non-uniformities in congestion — local differences in congestion may motivate *asymmetric* router designs, unlike the ones we have analyzed. Our initial experiments attempt to isolate the impact of backlogging on performance by using ideal token sinks at the output channels. However more realistic applications may experience backlogging at the destination ports which can cause further performance degradation. An appropriate study of backlogging would vary the amount of buffering on the interconnect channels. Buffering allows packets to be pushed further through the router during backlogging which frees up merge/split resources in earlier stages. Both critical channel analysis and temporal analysis on the FIFO occupancy can indicate whether channels have sufficient buffering to sustain reasonable performance.

This bit-router design example demonstrates the importance of studying execution details of different versions of a parallel program operating under different

workloads. Workloads and benchmarks can drive a significant portion of design choices and optimizations in large, complex designs. Without dynamic profiling, optimization choices are typically limited to those found by static analysis. Data from trace profiling enables designers to better assess design tradeoffs. The bit-router design is an example of an area-performance tradeoff study. Our analysis framework can be used to justify the selection of the best design, given the typical inputs and conditions under which it is expected to operate.

5.3 Summary

The examples in this chapter further demonstrate the capabilities of our trace analysis infrastructure. With fundamental trace analysis procedures, a designer can quickly discover the structural performance bottlenecks in an asynchronous circuit described in a high level. Our infrastructure provides not only a library of primitives and analysis procedures, but the ability to quickly develop new analysis routines and packages, tailored to specific needs of the designer. The Fibonacci example demonstrated how trace analyses can be written to guide the designer in the correct direction for optimizations, and the bit-router example demonstrated how analyses can be used to assess variations in design space exploration. Both of these examples are small for the purpose of proving the concept, however, the same analyses and summarizing procedures can be used to quickly reduce large volumes of data that arise from larger designs and longer traces. Finally, every designer may have different ways of diagnosing performance problems. Our trace analysis infrastructure does not restrict the ways in which queries and analyses can be constructed and chained together. It gives users absolute freedom and power to develop whatever analyses he or she can conjure.

CHAPTER 6

CONCLUSION

"It was on a dreary night of November that I beheld the accomplishment of my toils... I collected the instruments of life around me, that I might infuse a spark of being into the lifeless thing that lay at my feet... by the glimmer of the half-extinguished light, I saw the dull yellow eye of the creature open; it breathed hard, and a convulsive motion agitated its limbs. How can I describe my emotions at this catastrophe, or how delineate the wretch whom with such infinite pains and care I had endeavoured to form?"

Mary Wollstonecraft Shelley, *Frankenstein*

Asynchronous VLSI circuit design has been well understood for decades, however, mainstream adoption has been impeded by: shortage of educators and experts, long and incumbent legacy of traditional synchronous design, and skepticism and uninformed criticism of the asynchronous design. Nevertheless, asynchronous design is gradually becoming industrialized as designers realize the benefits it has to offer. Another frequently cited reason for slow pervasion of asynchronous circuit design is the shortage of design tools. This project was motivated by the need for asynchronous design tools, without which, design analysis and optimization would be extremely difficult and tedious.

Conventional approaches to asynchronous circuit synthesis usually start with a high level sequential (or concurrent) functional description of an application. Static program analysis and successive refinement (semantic-preserving decomposition and transformations) drive the synthesis of netlists in a top-down manner. However, there are numerous ways to decompose and synthesize asynchronous circuits, even within a single family such as QDI circuits. The overwhelming breadth of design and transformation choices may not be resolvable from static analysis alone. The role of *feedback* is to provide information from a latter phase of synthesis back up to a higher-level to better assess the impact of design choices and

with performance metrics, and steer synthesis in beneficial directions.

Figure 1.1 showed some steps in which profiling feedback can be used to improve synthesis. One application of particular interest is high-level program rewriting. By rewriting a concurrent program more explicitly, one can *restructure* circuits in ways that low-level, detailed synthesis cannot accomplish. High-level concurrent program specifications can express architectural organization more aptly than low-level netlists. Optimizing concurrent programs at a higher, structural level translates to better synthesized circuits. The task of writing a high-level, decomposed concurrent program specification for an asynchronous design is often assigned to a (preferably experienced) human, however, our work takes one more step towards completing the loop for automatic program rewriting and design space exploration: providing a versatile framework for evaluating high-level concurrent programs through simulation tracing and trace analysis.

Profiling is especially beneficial in applications where design decisions cannot be statically evaluated. Difficulties arise when a particular program transformation is non-obvious (perhaps due to complex interactions or timing), dependent on input data and characteristics, or involves some tradeoff between metrics. Profiling simulation traces of concurrent programs can give a designer or compiler an idea of the relative impact of a transformation, and most importantly, significantly prune the space of optimizing transformations to apply. Without profile-directed optimization, designers are left to guess or exhaustively explore innumerable variations of a design. We have shown in Chapter 4 how even simple transformations commonly used in asynchronous circuits can benefit from trace analyses. Not only is trace analysis useful for assisting human-interactive design iterations, it is *necessary* for automating high-level optimizations in future asynchronous design tools.

Our contribution towards the effort of automating high-level program rewriting and design space exploration consists of an asynchronous circuit compiler, simulator, and trace analysis framework, described in Chapter 3. Our analysis framework includes a Scheme environment that allows users to interactively mine detailed simulation traces for data that measures the impact of program transformations on performance (or other metric). The framework is built upon a library of primitive procedures (API) for working with the compiler's hierarchical object files and simulator's trace files. We have also developed a library of analysis procedures based on critical paths and path statistics, however, developers are free (and encouraged) to develop custom analysis packages using the provided extensible framework.

Our trace analysis infrastructure has been shown to be helpful in evaluating asynchronous circuits at a high-level of abstraction, which helps designers write more structurally optimized, high-level specifications. The implementation described herein is a prototype fragment for the development of future asynchronous design tools. Once static analysis and program rewriting are supported in the framework, and a design space exploration engine leverages the capabilities of profile-guided transformation and optimization, will we see a truly powerful and intelligent asynchronous circuit design compilers.

EPILOGUE

> **HAL**: *"Look, Dave, I can see you're really upset about this. I honestly think you ought to sit down calmly, take a stress pill, and think things over..."*
>
> **HAL**: *"I know I've made some very poor decisions recently, but I can give you my complete assurance that my work will be back to normal. I've still got the greatest enthusiasm and confidence in the mission. And I want to help you."*
>
> from *2001: A Space Odyssey*,
> by Arthur C. Clarke (1917–2008)

In his final year of graduate school, the author's dissertation writing was interrupted when he joined Achronix Semiconductor Corporation, an Asynchronous FPGA startup company founded by his academic siblings and advisor, to aid in the verification and completion of their first commercial chip. The dissertation was finished in the Spring of 2008 and defended on May 12th, 2008.

Upon completion of this dissertation, the author resumed his duties at Achronix where some of his asynchronous circuit tools are used for designing and simulating asynchronous circuits. All of the tools he has developed (HACKT) have been released under an open source license.

APPENDIX A

CHP QUICK REFERENCE

The CHP notation we use is based on Hoare's CSP [25]. A full description of CHP and its semantics can be found in [44]. What follows is a short and informal description.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := \mathit{true}$, and $a\downarrow$ for $a := \mathit{false}$.
- Selection: $[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$, where Gi's are boolean expressions (guards) and Si's are program parts. The execution of this command corresponds to waiting until one of the guards is *true*, and then executing one of the statements with a *true* guard. The notation $[G]$ is short-hand for $[G \rightarrow \mathit{skip}]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the *true* guards and executing the corresponding statement, repeating this until all guards evaluate to *false*. The notation $*[S]$ is short-hand for $*[\mathit{true} \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is *true* iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.

- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

APPENDIX B

CHP PROCESS LIBRARY

B.1 Buffers

Program B.1: `bool-buf` CHP process

```
defproc bool_buf (chan?(bool) L; chan!(bool) R) {
bool x;
chp {
  *[L?(x); R!(x)]
}
}
```

Program B.2: `bool-buf-init` CHP process

```
template <pbool B>
defproc bool_buf_init (chan?(bool) L; chan!(bool) R) {
bool x;
chp {
  x:=B;
  *[R!(x); L?(x)]
}
}
```

Program B.3: `bool-peekbuf` CHP process

```
defproc bool_peekbuf(chan?(bool) L; chan!(bool) R) {
bool x;
chp {
*[ L#(x); L?, R!(x) ]
}
}
```

B.2 Functions

Program B.4: `bool-and` CHP process

```
template <pint N>
defproc bool_and(chan?(bool) A[N]; chan!(bool) O) {
bool a[N];
chp {
  *[{,i:N: A[i]?(a[i])};  O!((&&:i:N: a[i]))]
```

```
}
}
```

Program B.5: `bool-table` CHP process

```
template <pint N; pbool V[N]>
defproc bool_lookup_table(chan?(int) A; chan!(bool) D) {
int a;
chp {
*[ A?(a); D!(V[a]) ]
}
}
```

B.3 Environments

Program B.6: `bool-sink` CHP process

```
defproc bool_sink(chan?(bool) B) {
chp {
  *[ B? ]
}
}
```

Program B.7: `bool-source` CHP process

```
template <><pint N; pbool B[N]>
defproc bool_source(chan!(bool) S) {
chp {
  *[ {;i:N: S!(B[i]) } ]
}
}
```

B.4 Flow Control

Program B.8: `bool-copy` CHP process

```
template <pint N>
defproc bool_copy (chan?(bool) A; chan!(bool) O[N]) {
bool a;
chp {
  *[ A?(a); {,i:N: O[i]!( a) } ]
}
}
```

Program B.9: `bool-merge` CHP process

```
template <pint N>
defproc bool_merge(chan?(int) C; chan?(bool) I[N]; chan!(bool) O) {
int c;
bool x;
chp {
*[ C?(c);
   I[c]?(x);
   O!(x)
]
}
}
```

Program B.10: `bool-split` CHP process

```
template <pint N>
defproc bool_split (chan?(int) C; chan?(bool) I; chan!(bool) O[N]) {
int c;
bool x;
chp {
*[ C?(c),I?(x);
   O[c]!(x)
]
}
}
```

B.5 Alternators

Program B.11: `bool-split-alternator` CHP process

```
template <pint N>
defproc bool_split_alternator(chan?(bool) I; chan!(bool) O[N]) {
bool v[N];
chp {
  *[ {;i:N: I?(v[i]);  O[i]!(v[i])  } ]
}
}
```

Program B.12: `bool-merge-alternator` CHP process

```
template <pint N>
defproc bool_merge_alternator(chan?(bool) I[N]; chan!(bool) O) {
bool v[N];
chp {
  *[ {;i:N: I[i]?(v[i]);  O!(v[i])  } ]
```

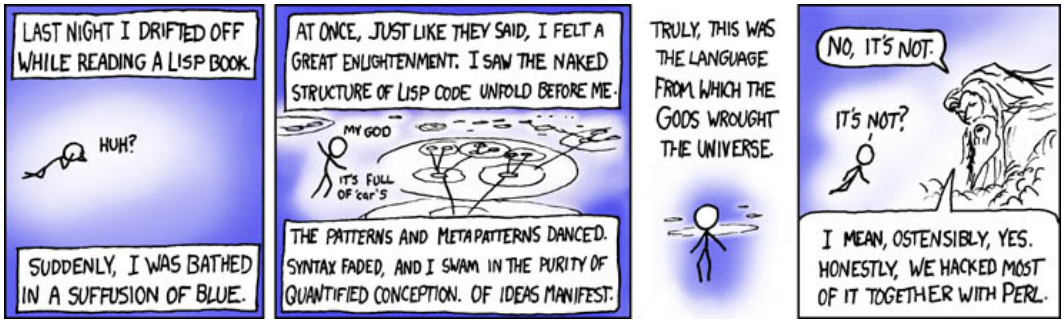
```
}
}
```

Program B.13: `bool-parallel-fifo` CHP process

```
template <pint N>
defproc bool_parallel_fifo (chan?(bool) I; chan!(bool) O) {
chan(bool) M[N];
 bool_split_alternator <N> s(I, M);
bool_merge_alternator<N> m(M, O);
}
```

APPENDIX C

SCHEME UTILITY LIBRARY



Randall Munroe, `http://www.xkcd.com/224/`

This appendix lists some common procedures used over the course analysis of development. Many are standard scheme library procedures. All of the procedures listed here are generic, that is, they are not specific to our own work. We provide the source for procedures that we defined.

C.1 Queues

Standard queue operations are provided by the `(ice-9 q)` Scheme module. We use the following functions in our libraries:

```
make-q q-empty? q-front q-rear q-push! q-pop!
```

C.2 Algorithms

We frequently used the following algorithms and higher-order procedures in our libraries:

```
for-each map filter partition find find-tail any
```

```
(define (forward-accumulate binop init lst)
  (if (null? lst) init
    (forward-accumulate binop
      (binop (car lst) init) (cdr lst))))
(define accumulate forward-accumulate)
```

```
(define (reverse-accumulate binop init lst)
  (if (null? lst) init
    (binop (car lst)
      (reverse-accumulate binop init (cdr lst)))))

(define (list-flatten lstlst)
"Converts a list-of-lists into a single flat list."
  (reverse-accumulate append '() lstlst))

(define (list-flatten-reverse lstlst)
; Converts a list-of-lists into a single
; flat list (reverse-constructed)."
  (accumulate append '() lstlst))

(define (filter-split pred? lst)
; Partitions a list into a pair of lists, the first
; of which satisfied the predicate, the second of
; which failed the predicate.
; NOTE: result of partition reverses list order.
  (receive (sat unsat)
    (partition pred? lst)
    (cons sat unsat)))

(define (find-assoc-ref alst key)
; Finds the key-value *pair* in an associative list
; using equal?, given a key.
; In contrast, assoc-ref returns only the value.
  (find (lambda (x) (equal? (car x) key)) alst))

(define (iterate-template prod op index inc term?)
; Iteration template, where @var{prod} is a
; cumulative value (may be object), @var{op} is
; the combining function operating on
; (@var{index}, @var{prod}), @var{index} is a
; counter, @var{inc} is an incrementing procedure,
; and @var{term?} is a termination predicate.
  (if (term? index) prod
    (iterate-template
      (op index prod) op (inc index) inc term?)))

(define (iterate-default prod op index limit)
; Iterate from @var{index} up to @{limit},
; incrementing.
  (iterate-template prod op index 1+
    (lambda (c) (> c limit))))
```

```
(define (iterate-reverse-default prod op index limit)
; Iterate from @var{index} down to
; @{limit}, decrementing.
  (iterate-template prod op index 1-
    (lambda (c) (< c limit))))
```

C.3 Red-black Trees

All of our associative and ordered maps use red-black trees for their self-balancing properties. Their description can be found in any standard text data structures. We only name the interface functions here for brevity.

`<rb-tree>` [Class]

Red-black tree data structure. Each tree uses two comparator functors, one for ordering, one for equality.

```
(define (rb-tree? t) (is-a? t <rb-tree>))
```

`<rb-tree-node>` [Class]

Tree node, which contains a key-value pair.

`(make-rb-tree` *`key=?  key<?`*`)` [Procedure]

Contruct an empty tree.

`(rb-tree/insert!` *`tree key value`*`)` [Procedure]

Insert *value* associated with *key*.

`(rb-tree/insert-if-new!` *`tree key value`*`)` [Procedure]

Only insert value if key didn't not exist before.

(rb-tree/delete! *tree key*) [Procedure]

(rb-tree/lookup-key *tree key default*) [Procedure]

(rb-tree/lookup *tree key default*) [Procedure]

(rb-tree/lookup-pair *tree key default*) [Procedure]

(rb-tree/lookup-mutate! *tree key proc-1 default*) [Procedure]

(rb-tree/copy *tree*) [Procedure]

(rb-tree/height *tree*) [Procedure]

(rb-tree/size *tree*) [Procedure]

(rb-tree/empty? *tree*) [Procedure]

(rb-tree/equal? *x y value=?*) [Procedure]

(rb-tree/for-each *pair-proc tree*) [Procedure]

(rb-tree/merge *tree1 tree2 combine-value*) [Procedure]

(rb-tree/intersect *tree1 tree2 combine-value*) [Procedure]

(rb-tree/map-pairs *tree pair-proc*) [Procedure]

(rb-tree->stream *tree*) [Procedure]

(rb-tree/key-list *tree*) [Procedure]

(rb-tree/value-list *tree*) [Procedure]

(rb-tree/min-key *tree default*) [Procedure]

`(rb-tree/min-value` *tree default*`)` [Procedure]

`(rb-tree/min-pair` *tree*`)` [Procedure]

Procedures for `max` are analogous to those of `min`.

`(rb-tree/increment!` *tree key*`)` [Procedure]

Increment counter by 1 or initialize to 1 if doesn't already exist.

We also provide interfaces to unordered associative lists (consisting of key-value pairs).

`(alist->rb-tree` *alist key=? key<?*`)` [Procedure]

`(rb-tree->alist` *tree*`)` [Procedure]

C.4 Streams

The following stream procedures are provided by the `(ice-9 streams)` module:

```
make-stream stream-car stream-cdr stream-null?
list->stream stream->list stream->reversed-list
vector->stream stream->vector
stream-fold stream-for-each stream-map
```

We provide the remaining stream procedures:

```
(define the-empty-stream '())

(define (cons-stream h t) (delay (cons h t)))
```

```
(define (nth-stream n s)
; References the Nth element of the stream.
  (if (= n 0) (stream-car s)
    (nth-stream (- n 1) (stream-cdr s))))
(define stream-ref nth-stream)

(define (stream-filter pred? stream)
; "Like the filter algorithm, but operating
; on a stream instead of a list.
  (cond ((stream-null? stream)
          (delay the-empty-stream))
  ((pred? (stream-car stream))
    (cons-stream (stream-car stream)
    (stream-filter pred?
            (stream-cdr stream))))
  (else (stream-filter pred?
          (stream-cdr stream)))))

(define (stream-filter-split pred? stream)
; partitions into two streams using predicate
  (if (stream-null? stream)
    (cons (delay the-empty-stream)
      (delay the-empty-stream))
    (let ((head (stream-car stream))
    (rem (stream-filter-split pred?
            (stream-cdr stream))))
  (if (pred? head)
    (cons (cons-stream head (car rem)) (cdr rem))
    (cons (car rem) (cons-stream head (cdr rem))))
    )))

(define (stream-start pred? stream)
; Truncates the stream up to the first element
; that satisfies the predicate.
  (cond ((stream-null? stream)
          (delay the-empty-stream))
  ((pred? (stream-car stream)) stream)
  (else (stream-start pred? (stream-cdr stream)))
  ))

(define (stream-stop pred? stream)
; Truncates the stream after the first element
; that satisfies the predicate.
  (cond ((stream-null? stream)
          (delay the-empty-stream))
```

```
  ((pred? (stream-car stream))
    (delay the-empty-stream))
  (else (cons-stream (stream-car stream)
    (stream-stop pred? (stream-cdr stream))))
  ))

(define (stream-crop p1 p2 stream)
; Truncates the stream until the first predicate
; is satisfied, then truncates the stream after
; the second predicate is satisfied.
  (stream-stop p2 (stream-start p1 stream)))

(define (stream-accumulate op initial stream)
; the accumulate algorithm for streams
  (if (stream-null? stream) initial
    (op (stream-car stream)
      (stream-accumulate op initial
        (stream-cdr stream)))))

(define (stream-concat s1 s2)
; Concatenates two streams, by exhausting the
; first stream first.
  (if (stream-null? s1) s2
     (cons-stream (stream-car s1)
       (stream-concat (stream-cdr s1) s2))))

(define (stream-flatten strstr)
; Flattens a stream of streams sequentially
; into a single concatenated stream.
  (stream-accumulate stream-concat
    the-empty-stream strstr))

(define (stream-of-lists->stream strlst)
; Flattens a stream of lists into a
; single concatenated stream.
  (stream-accumulate (lambda (x y)
    (stream-concat (list->stream x) y))
      (delay the-empty-stream) strlst))

; finite stream of integers
(define (enumerate-interval-stream low high)
; Generate a stream of integers from [low,high].
  (iterate-reverse-default (delay the-empty-stream)
    cons-stream high low))
```

```
; finite stream of integers, decreasing order
(define (enumerate-interval-reverse-stream low high)
; Generate a stream of integers from [high,low].
  (iterate-default (delay the-empty-stream)
    cons-stream low high))
```

APPENDIX D

HAC OBJECT FILE API

This appendix lists many of the Scheme functions that operate on a HAC object file to extract information from the hierarchical intermediate representation. Functions prefixed with '`hac:`' are primitives implemented in C++.

`(hac:objdump )` [Procedure]

print textual representation of entire object file.

`(hac:parse-reference` *`str`*`)` [Procedure]

Translates string *str* (as it would appear in source) to a global instance reference handle, and returns the handle.

`(hac:lookup-reference-aliases` *`ref`*`)` [Procedure]

Return a list of all equivalent names of instance reference *ref*.

`(hac:valid-process-id?` *`id`*`)` [Procedure]

Return `true` if *id* is a valid process instance number.

`(hac:reference-type` *`ref`*`)` [Procedure]

Returns a handle to the type associated with instance *ref*.

`(reference-equal?` *`r1 r2`*`)` [Procedure]

Return `true` if references *r1* and *r2* refer to the same instance.

`(process-id->string` *`pid`*`)` [Procedure]

Return a string of the canonical instance name refenced by global process number *pid*.

APPENDIX E

HAC CHP SIMULATOR STATE API

E.1 Event retrieval

Basic event lookup only requires a global event index, *eid*.

`(hac:chpsim-get-event` *`eid`*`)` [Procedure]

> Returns a pair, `(eid . #<chpsim-event>)`. The event object (at `cdr`) does not encode its own index number.

We use the following procedure aliases for clarity.

```
(define static-event-node-index car)
(define static-event-raw-entry cdr)
```

E.2 Event predicates

For the following primitive predicate functions, parameter *ev* is a `chpsim` event object in Scheme, `#<chpsim-event>`.

`(hac:chpsim-event-trivial?` *`ev`*`)` [Procedure]

> Returns true if event has no real action, e.g. concurrent forks, concurrent joins, end-of-selections.

`(hac:chpsim-event-wait?` *`ev`*`)` [Procedure]

> True if event is a condition wait.

`(hac:chpsim-event-assign?` *`ev`*`)` [Procedure]

> True if event is a value assignment, or $x := y$ in CHP.

`(hac:chpsim-event-send? ev)` [Procedure]

True if event is a channel send.

`(hac:chpsim-event-receive? ev)` [Procedure]

True if event is a channel receive.

`(hac:chpsim-event-peek? ev)` [Procedure]

True if event is a channel peek.

`(hac:chpsim-event-fork? ev)` [Procedure]

True if event is a channel fork.

`(hac:chpsim-event-join? ev)` [Procedure]

True if event is a channel join.

`(hac:chpsim-event-select? ev)` [Procedure]

True if event is a selection (choice), determinstic or nondeterministic, but excludes while-do branches.

`(hac:chpsim-event-select-det? ev)` [Procedure]

True if event is a deterministic selection.

`(hac:chpsim-event-select-nondet? ev)` [Procedure]

True if event is a nondeterministic selection.

`(hac:chpsim-event-branch? ev)` [Procedure]

True if event is any selection or while-do branch.

`(hac:chpsim-event-while-do? ev)` [Procedure]

True if event is while-do branch.

E.3 Event properties

`(hac:chpsim-event-process-id` *ev*`)` [Procedure]

Returns the global process index to which the event belongs.

`(hac:chpsim-event-delay` *ev*`)` [Procedure]

Returns the time delay associated with the event.

`(hac:chpsim-event-num-predecessors` *ev*`)` [Procedure]

Returns the number of predecessors that *must* arrive before this event may be checked for execution. Only concurrent join events have more than one required predecessor.

`(hac:chpsim-event-num-successors` *ev*`)` [Procedure]

Returns the number of successors (outgoing edges) that *may* follow this event. For concurrent forks, every successor is followed; for selections, only one successor branch is taken.

`(hac:chpsim-event-successors` *ev*`)` [Procedure]

Returns the list of successor events that may follow this event.

`(hac:chpsim-event-source` *ev*`)` [Procedure]

Print the full context of the CHP source that produced this event node.

`(hac:chpsim-event-may-block-deps-internal` *ev*`)` [Procedure]

Internal function. Produces a set of instance of references that this event *may* depend on to unblock; i.e. instances in this set are subscribed upon blocking.

E.4 Pre-computed static data

`chpsim-num-events` [Variable]

The number of events in the whole program, allocated by the simulator.

`all-static-events-stream` [Variable]

The set of *all* global events in the whole program. Defining as a stream leverages lazy evaluation and memoization, i.e., it is computed and cached upon first reference. The stream can be fed to arbitrary functions and filters for querying. (Section 3.4)

E.5 Static analysis routines

The following procedures are non-primitive routines defined in Scheme. We include the procedure source for some of them.

`(chpsim-event-may-block-deps` *ev*`)` [Procedure]

Returns the set of block dependencies in a more meaningful structure, used with `hac:chpsim-event-may-block-deps-internal`.

The following procedure filters out a stream of events given an predicate (expects an event object). *estrm* can be *all-static-events-stream* or any subset thereof.

Program E.1: `chpsim-filter-static-events` procedure

```
(define (chpsim-filter-static-events pred? estrm)
  (stream-filter
    (lambda (e) (pred? (static-event-raw-entry e)))
    estrm))
```

The next variation expects predicates that operate on (index,event) pairs:

Program E.2: `chpsim-filter-static-events-indexed` procedure

```
(define (chpsim-filter-static-events pred? estrm)
  (stream-filter (lambda (e) (pred? e))
    estrm))
```

E.5.1 Event graphs

This section contains listings for procedures that produce basic information about event graph connectivity. `chpsim-assoc-event-successors` (Program E.3) constructs adjacency lists for every node in the whole program event graph. The returned data structure is a stream of (index, list-of-index)-pairs. The result can be used to perform static graph analysis structures, such as dominator-trees and postdominator-trees.

Program E.3: `chpsim-assoc-event-successors` procedure

```
(define (chpsim-assoc-event-successors estrm)
  (stream-map (lambda (e)
    (cons (static-event-node-index e)
      (hac:chpsim-event-successors
        (static-event-raw-entry e))))
    estrm))
```

Section 3.4.1 discussed the use of memoized variable computations, listed here:

Program E.4: `static-event-successors-map-delayed` variable

```
(define static-event-successors-map-delayed
  (delay
    (let ((succs-map (make-rb-tree = <)))
      (stream-for-each
        (lambda (e) (rb-tree/insert! succs-map
          (successor-map-key e)
          (successor-map-value-list e)))
        (chpsim-assoc-event-successors
          all-static-events-stream))
      succs-map)))
```

Program E.5: `static-event-predecessors-map-delayed` variable

```
(define static-event-predecessors-map-delayed
  (delay
    (let ((preds (make-rb-tree = <)))
      (stream-for-each (lambda (e)
        (rb-tree/insert! preds e
          (make-rb-tree = <)))
        chpsim-static-event-index-stream)
      (rb-tree/for-each (lambda (x)
        (for-each (lambda (y)
          (rb-tree/lookup-mutate! preds y
            (lambda (z) (rb-tree/insert! z
              (successor-map-key x) '()) z) #f))
          (successor-map-value-list x)))
```

```
      (force static-event-successors-map-delayed))
    preds)))
```

`(chpsim-successor-lists->histogram` *succ-list*`)` [Procedure]

Constructs a zero-initialized histogram from a list of successor-adjacency lists. This is often called prior to accumulating statistics over successors taken. (Program E.6)

Program E.6: `chpsim-successor-lists->histogram` variable

```
(define (chpsim-successor-lists->histogram succ-list)
(let ((ret-histo (make-rb-tree = <)))
  (map (lambda (x)
    (rb-tree/insert! ret-histo (car x)
      (let ((sub-histo (make-rb-tree = <)))
        (for-each (lambda (y)
          (rb-tree/insert! sub-histo y 0)) (cdr x))
        sub-histo)))
    succ-list)
  ret-histo))
```

Program E.7: `static-events-with-multiple-entries-delayed` variable

```
(define static-events-with-multiple-entries-delayed
(delay
  (let ((ret-map (make-rb-tree = <))
        (preds (force
          static-event-predecessors-map-delayed)))
    (stream-for-each (lambda (e)
      (rb-tree/insert! ret-map
        (static-event-node-index e) '()))
      (stream-filter (lambda (x)
        (let ((i (static-event-node-index x))
              (e (static-event-raw-entry x)))
          (> (rb-tree/size
             (rb-tree/lookup preds i #f))
             (hac:chpsim-event-num-predecessors e))))
        all-static-events-stream))
    ret-map)))
```

E.5.2 Graph traversal

The following function performs a predicated depth-first traversal over the whole program event graph. A visited-set is maintained to ensure that no event is more than once. The function expects a procedure object, *thunk*, and an event predicate, *pred?*. (Discussed in Section 3.4.2)

Program E.8: `static-events-depth-first-walk-predicated` procedure

```
(define (static-events-depth-first-walk-predicated
           thunk pred?)
  (let ((visited (make-rb-tree = <))
        (succs-map (force
          static-event-successors-map-delayed)))
    (let loop ((n root-event-id))
      (if (not (rb-tree/lookup-key visited n #f))
        (begin
          (rb-tree/insert! visited n '())
          (thunk n)
          (if (pred? n)
            (for-each (lambda (s) (loop s))
              (rb-tree/lookup succs-map n #f))))
      ))))
```

The following depth-first traversal is written iteratively to take advantage of proper tail recursion, which works for any graph in a bounded stack space. This method is generally preferred where there are extremely long loops in the event graph. The algorithm works by passing and manipulating a two-level worklist of nodes to visit, which represents the execution stack in a recursive implementation. The 'breadcrumbs' trail represents the current stack, and is actually redundant with the two-level worklist. `thunk-node` is a procedure to execute upon visiting each node for the first time, and `thunk-back` is a procedure to execute upon discovering a back edge. The running time for this algorithm is actually $O(n^2)$ in the worst case because of the linear search through the 'breadcrumbs' stack to detect cycle-forming back edges.

Program E.9: `static-events-depth-first-walk-iterative` procedure

```
(define (static-events-depth-first-walk-iterative
    thunk-node thunk-back)
```

```
  (let ((visited (make-vector chpsim-num-events #f))
        (succs-map (force
          static-event-successors-map-delayed)))
    (let loop ((breadcrumbs '())
               (worklist (list (list root-event-id))))
      (if (not (null? worklist))
        (let ((l (car worklist))
              (r (cdr worklist)))
          (if (null? l)
            ; tail call, remove last predecessor
            (if (not (null? r))
              (loop (cdr breadcrumbs)
                (cons (cdar r) (cdr r))))
            ; else outgoing edge in last list
            (let ((n (car l)))
              (if (vector-ref visited n) (begin
                  ; then detect cycle
                  (if (any (lambda (x) (= x n))
                    breadcrumbs)
                      (thunk-back (cons
                        (caadr worklist) n)))
                  (loop breadcrumbs (cons (cdr l)
                    (cdr worklist))))
                ; else not already visited
                (begin
                  (vector-set! visited n #t)
                  (thunk-node n)
                  (loop (cons n breadcrumbs)
                    (cons (rb-tree/lookup
                      succs-map n #f)
                      worklist))
))))))))
```

E.5.3 Event loops

The following procedure finds all loop back edges in forever-loops and do-while loops (Section 3.4.1):

Program E.10: `static-loop-bound-events-delayed` variable

```
(define static-loop-bound-events-delayed
(delay
  (let ((succs-map (force
```

```
        static-event-successors-map-delayed))
      (loop-backs (make-rb-tree = <))
      (loop-heads (make-rb-tree = <)))
  (static-events-depth-first-walk-iterative
    (lambda (n) #f) ; do nothing
    (lambda (p)
      (let ((n (car p)) (s (cdr p)))
        (rb-tree/insert! loop-backs n s)
        (rb-tree/insert! loop-heads s n)
      )))
  (cons loop-heads loop-backs))))
```

The following delayed evaluation finds all back edges of only do-while loops in a single pass over all static events. The result is a forward mapping and reverse mapping of back edges to do-while events.

Program E.11: `static-do-while-bound-events-delayed` delayed variable

```
(define static-do-while-bound-events-delayed
(delay (let ((succs-map (force
        static-event-successors-map-delayed))
      (do-while-backs (make-rb-tree = <))
      (do-while-heads (make-rb-tree = <)))
  (stream-for-each (lambda (ev)
    (let* ((n (static-event-node-index ev))
           (this-succs (rb-tree/lookup
             succs-map n #f)))
      (for-each
        (lambda (s)
          (if (hac:chpsim-event-do-while?
                (static-event-raw-entry
                  (hac:chpsim-get-event s)))
            (begin
              ; invariant: head-tails are 1-1 mapping
              (rb-tree/insert! do-while-backs n s)
              (rb-tree/insert! do-while-heads s n))))
        this-succs)))
    all-static-events-stream)
  (cons do-while-heads do-while-backs))))
```

The following procedure just returns a filtered stream of do-while selection events.

Program E.12: `static-do-while-events-delayed` variable

```
(define static-do-while-events-delayed
  (delay (let ((do-whiles (make-rb-tree = <)))
    (stream-for-each
```

```
      (lambda (n)
        (let ((e (static-event-raw-entry n)))
          (if (hac:chpsim-event-do-while? e)
            (rb-tree/insert! do-whiles
        (static-event-node-index n) '())))))
      all-static-events-stream)
    do-whiles)))
```

E.5.4 Selection events

Branch heads and tails can be found with the following evaluations (from Section 3.4.1):

Program E.13: `static-branch-bound-events-delayed` variable

```
(define static-branch-bound-events-delayed
(delay (let ((preds-map (force
        static-event-predecessors-map-delayed))
      (branch-stack (make-q))
      (branch-heads (make-rb-tree = <))
      (branch-tails (make-rb-tree = <)))
  (static-events-depth-first-walk
    (lambda (n)
      (let ((e (static-event-raw-entry
             (hac:chpsim-get-event n))))
        (cond ((hac:chpsim-event-branch? e)
               (q-push! branch-stack n))
          ((and (> (rb-tree/size (rb-tree/lookup
                     preds-map n #f)) 1)
                (= (hac:chpsim-event-num-predecessors
                     e) 1)
                (not (q-empty? branch-stack))
                (not (hac:chpsim-event-do-while? e)))
            (let ((bh (q-pop! branch-stack)))
              (rb-tree/insert! branch-heads bh n)
              (rb-tree/insert! branch-tails n bh))
)))))
  (cons branch-heads branch-tails))))
```

E.5.5 Concurrent sections

Loop head and tail pairs can be found with the following evaluations (from Section 3.4.1):

Program E.14: `static-fork-join-events-delayed` variable definition

```
(define static-fork-join-events-delayed
(delay
  (let ((preds-map (force
          static-event-predecessors-map-delayed))
        (fork-stack (make-q))
        (fork-heads (make-rb-tree = <))
        (fork-joins (make-rb-tree = <)))
    (static-events-depth-first-walk
      (lambda (n)
        (let ((e (static-event-raw-entry
                   (hac:chpsim-get-event n))))
          (cond ((hac:chpsim-event-fork? e)
                 (q-push! fork-stack n))
            ((hac:chpsim-event-join? e)
              (let ((jh (q-pop! fork-stack)))
                (rb-tree/insert! fork-heads jh n)
                (rb-tree/insert! fork-joins n jh)
              ))
          ))))
    (cons fork-heads fork-joins))))
```

APPENDIX F

HAC SIMULATOR TRACE API

Many of the trace file functions are discussed in Section 3.5.

F.1 Primitives

Primitive trace file operations are all implemented in C++, but exported into the Scheme environment as procedures.

`(hac:dump-trace` *filename*`)` [Procedure]

Opens a trace file and prints a textual dump.

F.1.1 Event trace access

`(hac:open-chpsim-trace-accessor` *filename*`)` [Procedure]

Return a handle object for a trace file in random-access mode.

`(hac:open-chpsim-trace` *filename*`)` [Procedure]

Return a handle object for the named trace file in forward mode. This exists for efficiency reasons.

`(hac:open-chpsim-trace-reverse` *filename*`)` [Procedure]

Return a handle object for a trace file in reverse mode. This exists for efficiency reasons.

`(hac:chpsim-trace?` *trf*`)` [Procedure]

Return true if object is a forward mode trace file handle.

`(hac:chpsim-trace-reverse? trf)` [Procedure]

Return true if object is a reverse mode trace file handle.

`(hac:chpsim-trace-accessor? trf)` [Procedure]

Return true if object is a random-access mode trace file handle.

`(hac:chpsim-trace-valid? trf)` [Procedure]

Return true if forward mode trace file handle is in valid state.

`(hac:chpsim-trace-reverse-valid? trf)` [Procedure]

Return true if reverse mode trace file handle is in valid state.

`(hac:chpsim-trace-num-entries trf)` [Procedure]

Returns the number of events logged in the trace file. *trf* can be a forward or reverse mode trace handle object.

`(hac:current-trace-entry trf)` [Procedure]

Returns a tuple representing an entry in the event trace: trace index, timestamp, static event id, event-cause id. *trf* is a forward-mode trace file handle. Calling this also causes the event iterator to advance one position. The end-of-stream is signaled with a `null` object.

`(hac:current-trace-reverse-entry trf)` [Procedure]

Returns a tuple representing an entry in the event trace. *trf* is a reverse-mode trace file handle. Calling this also causes the event iterator to advance (retreat) one position.

`(hac:lookup-trace-entry trf ind)` [Procedure]

Returns a tuple representing an entry in the event trace, indexed *ind*. *trf* is a random-access mode trace file handle.

F.1.2 State trace access

The trail of variable value changes is kept separately from the event trace in the trace file. It is accessed with the following set of procedures.

`(hac:open-chpsim-state-trace` *filename*`)` [Procedure]

Opens the state change portion of the trace file, returns a handle object.

`(hac:chpsim-state-trace?` *trf*`)` [Procedure]

Return true if object is a value-trace trace file handle.

`(hac:chpsim-state-trace-valid?` *trf*`)` [Procedure]

Return true if value-state trace file handle is in valid state.

`(hac:current-state-trace-entry` *trf*`)` [Procedure]

Returns a tuple representing all variables that were changed by a single event: trace index, changed variables with new values. *trf* is value-trace file handle. Calling this also causes the event iterator to advance one position.

F.2 Procedures

F.2.1 Trace file operations

```
; Produces a stream of event trace entries
; in forward order.
; trace-stream is a forward-mode trace handle.
(define-public (make-chpsim-trace-stream trace-stream)
  (make-stream (lambda (s)
    (let ((p (hac:current-trace-entry s)))
      (if (null? p) '() (cons p s))))
    trace-stream))
```

```
; Produces a stream of event trace entries
; in reverse order.
; trace-stream is a reverse-mode trace handle.
(define-public (make-chpsim-trace-reverse-stream
    trace-stream)
  (make-stream (lambda (s)
    (let ((p (hac:current-trace-reverse-entry s)))
      (if (null? p) '() (cons p s))))
    trace-stream))

; Produces a stream of value-change entries.
(define-public (make-chpsim-state-trace-stream
    trace-stream)
  (make-stream (lambda (s)
    (let ((p (hac:current-state-trace-entry s)))
      (if (null? p) '() (cons p s))))
    trace-stream))
```

For convenience, the following procedures just open named trace files and produce event and state streams:

```
(define-public (open-chpsim-trace-stream tf)
  (make-chpsim-trace-stream
    (hac:open-chpsim-trace tf)))
(define-public (open-chpsim-trace-reverse-stream tf)
  (make-chpsim-trace-reverse-stream
    (hac:open-chpsim-trace-reverse tf)))
(define-public (open-chpsim-state-trace-stream tf)
  (make-chpsim-state-trace-stream
    (hac:open-chpsim-state-trace tf)))
```

For clarity, the following structure member accessors are provided:

```
; event trace index
(define-public (chpsim-trace-entry-index e)
  (car e))
; event timestamp
(define-public (chpsim-trace-entry-time e)
  (cadr e))
; static global event index
(define-public (chpsim-trace-entry-event e)
  (caddr e))
; critical event (trace index)
(define-public (chpsim-trace-entry-critical e)
  (cdddr e))
```

F.2.2 State change traces

The following procedures access fields of the state-change trace structures. The set of changed variables (and their values) are arranged by type. Some of these procedures are described in Section 3.5.3.

```
; Extracts the state-trace entry's event index.
(define-public (chpsim-state-trace-entry-index s)
  (car s))
; Extract value changes of a certain type (tag).
(define-public (chpsim-state-trace-entry-subset s tag)
  (list-ref s (1+ (type-tag->offset tag))))

; Extracts modified bool variables from state-trace.
(define-public (chpsim-state-trace-entry-bools s)
  (chpsim-state-trace-entry-subset s 'bool))
(define-public (chpsim-state-trace-entry-ints s)
  (chpsim-state-trace-entry-subset s 'int))
(define-public (chpsim-state-trace-entry-enums s)
  (chpsim-state-trace-entry-subset s 'enum))
(define-public (chpsim-state-trace-entry-channels s)
  (chpsim-state-trace-entry-subset s 'channel))
```

Given a complete history of all value changes, we can focus on any single variable.

Program F.1: `chpsim-state-trace-filter-reference`: Procedure to filter a state-change stream with only events that affect a single variable (type, index) pair

```
(define-public (chpsim-state-trace-filter-reference
    s rpair)
; Filters only state change entries that affect the
; variable referenced by rpair, a type-index pair.
  (stream-filter (lambda (t)
    (any (lambda (e)
      (reference-equal? (car e) rpair))
        (chpsim-state-trace-entry-subset t
          (reference-type rpair))))
  s))
```

Program F.2: `chpsim-state-trace-focus-reference`: Procedure to focus state-change on only the referenced variable, stripping away the unreferenced variables that change on the same events

```
; Re-structures the stream of value changes to focus
; on only the referenced variable.
(define-public (chpsim-state-trace-focus-reference
    s rpair)
  (stream-map (lambda (x)
    (cons (chpsim-state-trace-entry-index x)
      (let ((p (filter (lambda (e)
                 (reference-equal? (car e) rpair))
               (chpsim-state-trace-entry-subset x
                 (reference-type rpair)))))
        (if (null? p) '() (car p)))))
  s))
```

Program F.3: `chpsim-state-trace-single-reference-values`: Procedure to strip away the variable index from a focused state-change stream, leaving only event-index and value

```
; Produces a stream of (trace index, variable value)
; pairs that pertain to a single referenced variable.
(define (chpsim-state-trace-single-reference-values
    s rpair)
  (stream-map (lambda (e) (cons (car e) (cddr e)))
    (chpsim-state-trace-focus-reference s rpair)))
```

F.2.3 Critical path

The following procedures are the basis for critical path analysis, described in Section 3.6.

```
; (private)
; With a random-access trace file handle, return the
; trace index of the previous critical event.
(define (chpsim-trace-critical-path-iterator
    rand-trace entry)
  (hac:lookup-trace-entry rand-trace
    (chpsim-trace-entry-critical entry)))
```

Program F.4: `chpsim-trace-critical-path-from`: Procedure for extracting a critical path (stream) given a random-access event trace handle and a starting event index

```
; Construct a critical path event stream (backwards)
; from a given trace index.
```

```
; @var{rand-trace} is a random-access trace handle.
(define-public (chpsim-trace-critical-path-from
    rand-trace ev)
  (make-stream (lambda (s)
    (if (null? s) '()
      (let ((ci (chpsim-trace-critical-path-iterator
             rand-trace s)))
        ; only self-referential trace index is 0
        (if (= (chpsim-state-trace-entry-index s)
            (chpsim-trace-entry-index ci))
            (cons s '())
    (cons s ci)))))
    (hac:lookup-trace-entry rand-trace ev)))
```

Program F.5: `chpsim-trace-critical-path`: Combined procedure for opening an event trace, and extracting the critical path, starting from the last event

```
; Produces a stream representation of the critical
; path starting from the last event in the named
; trace file.
(define-public (chpsim-trace-critical-path tr-name)
  (chpsim-trace-critical-path-from
    (hac:open-chpsim-trace-accessor tr-name)
    (1- (hac:chpsim-trace-num-entries tr-name))))
```

The following procedure constructs a critical event histogram from the critical path, by counting occurrences of adjacent event pairs (i, j) in the critical event stream where event i is critical to j. The result is a sparse, ordered tree of trees with histogram counts.

Program F.6: `make-event-adjacency-histogram`: Procedure for constructing an adjacency histogram given a stream of critical events

```
(define-public (make-event-adjacency-histogram
    crit-stream)
  (define (init-bin key)
    (let ((tree (make-rb-tree = <)))
      (rb-tree/insert! tree key 1)
      tree))
  (let ((edge-histo (make-rb-tree = <))
        (crit-ev (stream-map
          chpsim-trace-entry-event crit-stream)))
    (stream-for-each
      (lambda (event cause)
```

```
          (let ((n (rb-tree/insert-if-new! edge-histo
                  event (init-bin cause))))
            (if (not (unspecified? n))
              ; then we found previous entry
              ; n is a tree of \[cause, count\] pairs
              (let ((c (rb-tree/insert-if-new!
                     n cause 1)))
                (if (not (unspecified? c))
                  (rb-tree/lookup-mutate!
                    n cause 1+ #f))))))
          crit-ev
        (stream-cdr crit-ev))
      edge-histo))

; just an alias
(define-public make-critical-event-histogram
  make-event-adjacency-histogram)
```

F.2.4 Branch statistics

The following procedure is used to construct a histogram of taken branches of selections (including while-do loops) from an event trace stream. (Described in Section 3.5.4)

Program F.7: `make-select-branch-histogram`: Procedure to construct a histogram of successors taken per branch

```
(define-public (make-select-branch-histogram
    trace-stream)
(let* ((select-succ-lists
         (chpsim-assoc-event-successors (force
           static-events-selects-stream-delayed)))
       (ll-histo (chpsim-successor-lists->histogram
         (stream->list select-succ-lists)))
       (sorted-assoc-pred
         (let ((pred-map (make-rb-tree = <)))
           (stream-for-each
             (lambda (s) (rb-tree/insert!
               pred-map (car s) (cdr s)))
             (stream-of-lists->stream
               (chpsim-assoc-event-pred-from-succ
                 select-succ-lists)))
```

```
          pred-map)))
  ; incrementing counter
  (define (count-selects x)
    (let ((f (rb-tree/lookup sorted-assoc-pred
               x #f)))
      (if f (let ((y (rb-tree/lookup ll-histo f #f)))
        (rb-tree/lookup-mutate! y x 1+ #f)))))
  (stream-for-each (lambda (e)
      (count-selects (chpsim-trace-entry-event e)))
    trace-stream)
  ll-histo))
```

F.2.5 Loop statistics

The following procedure is used to construct a histogram of loop iteration counts from an event trace stream. (Described in Section 3.5.4)

Program F.8: `make-loop-histogram`: Procedure to construct a histogram of loop occurrences

```
(define-public (make-loop-histogram
    trace-stream)
  (let ((loop-heads
          (force static-loop-head-events-delayed))
        (loop-histo (make-rb-tree = <)))
    (rb-tree/for-each (lambda (p)
      (rb-tree/insert! loop-histo (car p) 0))
      loop-heads)
    (stream-for-each (lambda (e)
      (let ((eid (chpsim-trace-entry-event e)))
        (if (chpsim-event-loop-head? eid)
          (rb-tree/lookup-mutate!
            loop-histo eid 1+ #f))))
      trace-stream)
    loop-histo))
```

F.2.6 Channel statistics

The following procedures for analyzing channels on critical paths are introduced in Section 3.6.2.

`make-critical-channel-event-pairs-list` takes a critical event stream with a channel index parameter and returns a list of send-receive event-pairs in the trace that occurred on the critical path. Recall that the critical path is listed in reverse-chronological order. The resulting stream will be in forward-chronological order. Each list element is a list of 1 or two events. Atomic send-receive events on the critical path are paired together, while other non-paired send-receive events are singleton elements.

Program F.9: `make-critical-channel-event-pairs-list`: Procedure to fold and filter channel events from a critical path

```
(define (make-critical-channel-event-pairs-list
    crit-stream cid)
(let* ((cev (alist->rb-tree (stream->list
         (stream-map (lambda (e) (cons (car e) #t))
           (chpsim-find-events-involving-channel-id
             cid all-static-events-stream))) = <))
       (sr-estrm (stream-filter
         (lambda (e) (rb-tree/lookup cev
           (chpsim-trace-entry-event e) #f))
         crit-stream)))
  (stream-fold (lambda (elem init)
    ; init will accumulate as a list
    (if (null? init)
      (list (list elem))
      (let ((recent (car init))
            (rest (cdr init)))
        (if (= (length recent) 2)
          ; last event already paired
          (cons (list elem) init)
          (let* ((prev (car recent))
                 (crit-prev
                   (chpsim-trace-entry-critical
                     prev)))
            (if (and (= (chpsim-trace-entry-index
                          elem) crit-prev)
                     (= (1+ crit-prev)
                       (chpsim-trace-entry-index prev))
                     (not (=
                       (chpsim-trace-entry-event elem)
                       (chpsim-trace-entry-event prev)
```

```
                    )))
                (cons (cons elem recent) rest) ; pair up
                (cons (list elem) init)))))))
      '() sr-estrm)))
```

`filter-critical-send-receive-pairs-list` filters out a list of send-receive event pairs from the previous procedure, dropping the singleton events.

Program F.10: `filter-critical-channel-event-pairs-list`: Filter to keep only paired send-receive channel events

```
(define (filter-critical-send-receive-pairs-list
    unfiltered-list)
  (filter (lambda (x) (= (length x) 2))
    unfiltered-list))
```

`make-critical-send-receive-pairs-list` is just a composition of the previous two procedures, producing a list of only critical send-receive event pairs.

```
(define (make-critical-send-receive-pairs-list
    crit-stream cid)
  (filter-critical-send-receive-pairs-list
    (make-critical-channel-event-pairs-list
      crit-stream cid)))
```

`count-send-receive-criticality` takes a list of send-receive critical event pairs, and returns a pair of counters, where the first value is the number of times sender was critical, and the second value is the number of times the receiver was more critical.

Program F.11: `count-send-receive-criticality`: Procedure to count occurrences of sender or receiver criticality

```
(define (count-send-receive-criticality lst)
  (fold (lambda (elem init)
    (let ((ev (cdr (hac:chpsim-get-event
            (chpsim-trace-entry-event (car elem))))))
      ; check the first element of each
      ; send-receive pair: which is more critical?
      (cond
        ((hac:chpsim-event-send? ev)
            (cons (1+ (car init)) (cdr init)))
        ((hac:chpsim-event-receive? ev)
            (cons (car init) (1+ (cdr init))))
        (else init) ; doesn't count
      )))
  '(0 . 0) lst))
```

The following procedure is merely a convenient composition of the above.

Program F.12: `channel-send-receive-criticality`: Composed procedure to count occurrences of sender or receiver criticality

```
(define (channel-send-receive-criticality
    crit-stream cname)
  (count-send-receive-criticality
    (make-critical-send-receive-pairs-list
      crit-stream
      (cdr (hac:parse-reference cname)))))
```

F.2.7 Process statistics

Program F.13: `make-critical-process-histogram`: Procedure to identify which processes the critical path lies in

```
(define (make-critical-process-histogram crit-path)
  (let ((proc-histo (make-rb-tree = <)))
    (stream-for-each
      (lambda (pid) (rb-tree/increment!
        proc-histo pid))
      (stream-map
        (lambda (e)
          (hac:chpsim-event-process-id
            (static-event-raw-entry
              (hac:chpsim-get-event
                (chpsim-trace-entry-event e)))))
      crit-path))
    proc-histo))
```

Program F.14: `print-named-critical-process-histogram`: Print name of process along with index and number of occurrences on the critical path from the given histogram

```
(define (print-named-critical-process-histogram
    proc-histo)
  (rb-tree/for-each
    (lambda (p)
      (display (process-id->string (car p)))
      (display ": ")
      (display p)
      (newline))
  proc-histo))
```

BIBLIOGRAPHY

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] Cedell Alexander, Donna Reese, and James C. Harden. Near-critical path analysis of program activity graphs. In *Proc. 2nd Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 308–317, 1994.

[3] Cedell A. Alexander, Donna S. Reese, James C. Harden, and Ron B. Brightwell. Near-critical path analysis: A tool for parallel program optimization. In *Proc. of the First Southern Symposium on Computing*, December 1998.

[4] P. C. Bates and W. C. Wilden. EDL: a basis for distributed system debugging tools. In *Proc. 15th Annual Hawaii Int'l Conf. on System Sciences*, pages 86–93, January 1982.

[5] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Proceedings of the 5th Conference on Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[6] Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. Technical Report CS-5253:TR:87, California Institute of Technology, 1988.

[7] Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 99–116. North-Holland, 1988.

[8] Maria Calzarossa, Luisa Massari, Alessandro P. Merlo, and Daniele Tessera. Parallel performance evaluation: The MEDEA tool. In *HPCN Europe 1996: Proc. of the Int'l Conf. and Exhibition on High-Performance Computing and Networking*, pages 522–529, London, UK, 1996. Springer-Verlag.

[9] Bernard Cole. Will self-timed asynchronous logic rescue CPU design?, August 2002. `http://www.fulcrummicro.com/press_archives/embedded_02-0824.pdf`.

[10] J. Cortadella, M. Kishinevsky, A.Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of

asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[11] T. Delaitre, M. J. Zemerly, P. Vekariya, G. R. Justo, J. Bourgeois, F. Schinkmann, F. Spies, S. Randoux, and S. C. Winter. *EDPEPPS: A Toolset for the Design and Performance Evaluation of Parallel Applications*, volume 1470 of *Lecture Notes in Computer Science*, pages 113–125. Springer-Verlag Berlin / Heidelberg, 1998.

[12] Doug Edwards and Andrew Bardsley. Balsa: an asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[13] Doug Edwards, Andrew Bardsley, Lilian Janin, Luis Plana, and Will Toms. *Balsa: a Tutorial Guide*, 2006. `ftp://ftp.cs.man.ac.uk/pub/amulet/balsa/3.5/BalsaManual3.5.pdf`.

[14] Virantha Ekanayake, Clinton Kelly IV, and Rajit Manohar. An ultra-low-power processor for sensor networks. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[15] Antonio Espinosa, Tomàs Margalef, and Emilio Luque. Automatic detection of parallel program performance problems. In *VECPAR*, pages 365–377, 1998.

[16] Antonio Espinosa, Tomàs Margalef, and Emilio Luque. Automatic performance evaluation of parallel programs. In *Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing (PDP '98)*, pages 43–49, 1998.

[17] David Fang. Width-adaptive and non-uniform access asynchronous register files. Master's thesis, Cornell University, December 2003.

[18] David Fang and Rajit Manohar. Non-uniform access asynchronous register files. In *Proceedings of the 10th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Hersonissos, Crete, April 2004.

[19] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proceedings of the 1998 International Conference on Computer Design*, pages 247–252, Austin, TX, October 1998.

[20] S. B. Furber, J. D. Garside, S. Temple, P. Day, and N. C. Paver. AMULET2e:

An asynchronous embedded controller. In *Proceedings of the 3rd Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299, April 1997.

[21] G. Goldszmidt, S. Katz, and S. Yemini. Interactive blackbox debugging for concurrent languages. *SIGPLAN Not.*, 24(1):271–282, 1989.

[22] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[23] Guile: Project GNU's extension language. `http://www.gnu.org/software/guile/`.

[24] Olav Hansen and Peter Fritzson. A performance analyzer for a parallel real-time functional language. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, pages 479–488, Washington, DC, USA, 1996. IEEE Computer Society.

[25] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[26] C. A. R. Hoare. Communicating sequential processes, June 2004. `http://www.usingcsp.com/cspbook.pdf`.

[27] Jeffrey K. Hollingsworth, R. Bruce Irvin, and Barton P. Miller. The integration of application and system based metrics in a parallel program performance tool. In *PPOPP '91: Proceedings of the third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 189–200, New York, NY, USA, 1991. ACM Press.

[28] Jeffrey K. Hollingsworth and Barton P. Miller. Parallel program performance metrics: A comparison and validation. In *Supercomputing*, pages 4–13, 1992.

[29] Xiandeng Huang and Christoph Steigner. A model-driven tool for performance measurement and analysis of parallel programs. In *HPCN Europe '95: Proc. of the Int'l Conf. and Exhibition on High-Performance Computing and Networking*, pages 612–617, London, UK, 1995. Springer-Verlag.

[30] C. Kelly IV, V. Ekanayake, and R. Manohar. SNAP: A sensor network asyn-

chronous processor. In *Proceedings of the 9th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, May 2003.

[31] Hans Jacobson, Erik Brunvand, Ganesh Gopalakrishnan, and Prabhakar Kudva. High-level asynchronous system design using the ack framework. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 93, Washington, DC, USA, 2000. IEEE Computer Society.

[32] Lilian Janin, A. Bardsley, and D. Edwards. Simulation and analysis of synthesised asynchronous circuits. *International Journal of Simulation Systems, Science and Technology*, 4(3-4):31–43, September 2003.

[33] P. A. Karlsen and P. T. Røine. A timing verifier and timing profiler for asynchronous circuits. In *Proceedings of the 5th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 13–23, 1999.

[34] Seon Wook Kim, Insung Park, and Rudolf Eigenmann. A performance advisory tool for novice programmers in parallel computing. In *Proc. Workshop on Language and Compilers for Parallel Computing*, 2000.

[35] Michael Kishinevsky, Jordi Cortadella, and Alex Kondratyev. Asynchronous interface specification, analysis and synthesis. In *Proceedings of the 35th Design Automation Conference (DAC '98)*, pages 2–7, New York, NY, USA, 1998. ACM Press.

[36] A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, and L. Lavagno. The use of Petri Nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems, and Computers*, 8(1):67–118, 1998.

[37] Kuan-Jen Lin and Chen-Shang Lin. Automatic synthesis of asynchronous circuits. In *Proceedings of the 28th Design Automation Conference (DAC '91)*, pages 296–301, New York, NY, USA, 1991. ACM Press.

[38] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995.

[39] Gordon Lyon, Robert Snelick, and Raghu Kacker. Synthetic-perturbation tuning of mimd programs. *J. Supercomputing*, 8(1):5–28, 1994.

[40] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Perfor-

mance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, Cancun, Mexico, April 1994.

[41] R. Manohar, Tak-Kwan Lee, and A. J. Martin. Projection: a synthesis technique for concurrent systems. In *Proceedings of the 5th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 125–134, April 1999.

[42] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Fourth International Conference on the Mathematics of Program Construction, Lecture Notes in Computer Science*. Springer-Verlag, July 1998.

[43] A. J. Martin, M. Nyström, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E.-V. Talvala, J. T. Tong, and A. Tura. The Lutonium: A sub-nanojoule asynchronous 8051 microcontroller. In *Proceedings of the 9th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 14, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, December 1986.

[45] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communications*, The UT Year of Programming Series. Addison-Wesley, 1990.

[46] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Proceedings of the Conference on Advanced Research in VLSI*, pages 351–373. MIT Press, 1991.

[47] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri V. Cummings, and Tak Kwan Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[48] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[49] Barton P. Miller, Morgan Clark, Jeffrey K. Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, 1990.

[50] Barton P. Miller, Jeffrey K. Hollingsworth, and Mark D. Callaghan. The paradyn parallel performance tools and PVM. Technical Report CS-TR-1994-1240, Univ. Madison-Wisconsin, 1994.

[51] Bernd Mohr. Performance evaluation of parallel programs in parallel and distributed systems. In *Conference on Algorithms and Hardware for Parallel Processing*, pages 176–187, 1990.

[52] Bernd Mohr. Simple: a performance evaluation tool environment for parallel and distributed systems. In *EDMCC2: Proceedings of the 2nd European Conference on Distributed Memory Computing*, pages 80–89, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[53] S. F. Nielsen, J. Sparso, and J. Madsen. Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation. In *Digital System Design, 2004. Euromicro Symposium on*, pages 298–305, August 2004.

[54] Cherri M. Pancake. Applying human factors to the design of performance tools. In *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 44–60, London, UK, 1999. Springer-Verlag.

[55] Ad Peeters and Mark de Wit. *Haste Manual*, 2005. `http://www.handshakesolutions.com/More_information/Downloads/Article-14902.html`.

[56] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated synthesis for asynchronous FPGAs. In *13th ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.

[57] Piyush Prakash and Alain J. Martin. Slack matching quasi delay-insensitive circuits. In *Proceedings of the 12th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2006.

[58] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[59] Hans Scholten and John Posthuma. A debugging tool for distributed systems. In *TENCON '93. Proc. Conference on Computer, Communication, Control and Power Engineering*, volume 1, pages 173–176, October 1993.

[60] Kamel Slimani, Yann Rémond, Gilles Sicard, and Marc Renaudin. *TAST Profiler and Low Energy Asynchronous Design Methodology*, volume 3254/2004, pages 268–277. Springer Berlin / Heidelberg, 2004.

[61] Robert Snelick. S-check: a tool for tuning parallel programs. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 107–112, Washington, DC, USA, 1997. IEEE Computer Society.

[62] Robert Snelick, Joseph Já Já, Raghu Kacker, and Gordon Lyon. Synthetic-perturbation techniques for screening shared memory programs. *Softw. Pract. Exper.*, 24(8):679–701, 1994.

[63] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Trans. Comput. Syst.*, 6(2):157–195, 1988.

[64] Handshake Solutions. TiDE: Timeless design environment, white paper. Technical report, Handshake Solutions, 2007.

[65] Ivan Sutherland, Bob Sproull, and David Harris. *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[66] F. te Beest, M. Verra, A. Peeters, M. Wit, and E. Woutersen. *Handshake Solutions Design Flow Manual*, October 2005.

[67] John Teifel, David Fang, David Biermann, Clinton Kelly IV, and Rajit Manohar. Energy-efficient pipelines. In *Proceedings of the 8th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Manchester, UK, April 2002.

[68] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proceedings of the 10th Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Hersonissos, Crete, April 2004.

[69] G. K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods. Occam: an asynchronous hardware description language? In *EUROMICRO 97. New*

Frontiers of Information Technology. Proceedings of the 23rd EUROMICRO Conference, pages 249–256, September 1997.

[70] Georgios K. Theodoropoulos. Building Parallel Distributed Models for Asynchronous Computer Architectures. In *Proceedings of the World Transputer Congress 1994 (Lake Como, Italy) with J.V.Woods*, pages 285–301. IOS Press, September 1994.

[71] Product brief: Timeless design environment (TiDE), January 2007. `http://www.handshakesolutions.com/assets/downloadablefile/leaflet_TiDE-v4-12874.pdf`.

[72] H. Truong and T. Fahringer. SCALEA: A performance analysis tool for distributed and parallel programs. In *Euro-Par 2002*, 2002.

[73] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 384–389, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[74] Wikipedia. Scheme (programming language), 2007. `http://en.wikipedia.org/wiki/Scheme_(programming_language)`.

[75] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, May 1991.

[76] Felix Wolf and Bernd Mohr. *Automatic Performance Analysis of MPI Applications Based on Event Traces*, volume 1900 of *Lecture Notes in Computer Science*, pages 123–132. Springer-Verlag Berlin / Heidelberg, Munich, Germany, August 2000.

[77] Catherine G. Wong and Alain J. Martin. Data-driven process decomposition for circuit synthesis. In *Proc. of the IEEE Conference on Electronic Circuits and Systems*, 2001.

[78] Catherine G. Wong and Alain J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Proceedings of the 40th Design Automation Conference (DAC '03)*, 2003.

INDEX