UNIVERSITY OF CALIFORNIA
Santa Barbara

Hierarchical Transactions for Hardware/Software Cosynthesis

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

by

Kunal Arya

Committee in Charge:
Professor Forrest Brewer, Chair
Professor Timothy Sherwood
Professor Li-C. Wang
Professor Tevfik Bultan

December 2014

The Dissertation of
Kunal Arya is approved:

Professor Timothy Sherwood

Professor Li-C. Wang

Professor Tevfik Bultan

Professor Forrest Brewer, Committee Chair

September 2014

Hierarchical Transactions for Hardware/Software Cosynthesis

by

Kunal Arya

Abstract

Hierarchical Transactions for Hardware/Software Cosynthesis

Kunal Arya

Modern heterogeneous devices provide of a variety of computationally diverse components holding tremendous performance and power capability. Hardware-software cosynthesis offers system-level synthesis and optimization opportunities to realize the potential of these evolving architectures. Efficiently coordinating high-throughput data to make use of available computational resources requires a myriad of distributed local memories, caching structures, and data motion resources. In fact, storage, caching, and data transfer components comprise the majority of silicon real estate. Conventional automated approaches, unfortunately, do not effectively represent applications in a way that captures data motion and state management which dictate dominant system costs. Consequently, existing cosynthesis methods suffer from poor utility of computational resources. Automated cosynthesis tailored towards memory-centric optimizations can address the challenge, adapting partitioning, scheduling, mapping, and binding techniques to maximize overall system utility.

This research presents a novel hierarchical transaction model that formalizes state and control management through an abstract data/control encapsulation semantic. It is designed from the ground-up to enable efficient synthesis across heterogeneous system components, with an emphasis on memory capacity constraints. It intrinsically encourages a high degree of concurrency and latency tol-

erance, and provides verification tools to ensure correctness. A unique data/execution hierarchical encapsulation framework guarantees scalable analysis, supporting a novel concept of state and control mobility. A front-end language allows concise expression of designer intent, and is structured with synthesis in mind. Designers express families of valid executions in a minimal format through high-level dependencies, type systems, and computational relationships, allowing synthesis tools to manage lower-level details. This dissertation introduces and exercises the model, discussing language construction, demonstrating control and data-dominated applications, and presenting a synthesis path that exhibits near-linear scalability with problem size.

To my family, for their love, support, and joy. To Hinal, who always managed to put a smile on my face. To my friends who kept me in good spirits and (perhaps more importantly) kept me on my toes. To Forrest, for sharing his enthusiasm, consideration, time, and singularly unique approach to engineering.

Contents

List of Figures

List of Acronyms

LT	Latency Tolerant/Tolerance
RTL	Register Transfer Logic
FPGA	Field-Programmable Gate Array
HTL	Hierarchical Transaction Language
HTSS	Hierarchical Transaction Simulator and Synthesizer
SoC	System-on-a-Chip
RSoC	Reconfigurable System-on-a-Chip
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
GPGPU	General Purpose Graphics Processing Unit
DSP	Digital Signal Processor
ABI	Application Binary Interface
RAM	Random Access Memory
BRAM	Block RAM
CDFG	Control/Data-flow Graph
TLM	Transaction-Level Modeling
KPN	Kahn Process Network
AST	Abstract Syntax Tree

Acknowledgements

We thank the National Science Foundation for their generous funding. We also thank Intel, along with the Silicon Research Corporation, through whom we received custom funding into latency tolerant synthesis (the foundation of this work).

Chapter 1

Introduction

Heterogeneous hardware/software cosynthesis is the automated binding, allocation, and scheduling of an application onto hardware and software components. The target architecture spans the spectrum across arithmetic units, microprocessors, memory hierarchies/allocations, and the myriad bus/communication interfaces that allow them to execute cooperatively. This dissertation introduces the *hierarchical transactional semantic*, an abstract application specification model designed from the ground up to target cosynthesis in a way that balances high-level behavior abstraction with high performance synthesis. This balance is achieved through a novel hierarchical data/control encapsulation semantic, encouraging architectural exploration through tool-accessible flexibility in how state and execution are realized. With any abstract specification model comes the risk of imposing performance-hindering constraints, such as handshaking or tight synchronization mechanisms that prevent exploitation of parallel performance. The model, however, enables scalable synthesis, bounded analysis, clean and concise design specification, and verification without sacrificing performance. Finally, the management of memory and memory capacity is crucial to meeting high perfor-

mance implementations. The presented semantics allow direct control over data motion, capturing cost models that genuinely reflect their underlying architecture.

The evolution of computationally diverse heterogeneous platforms has opened new opportunities to meet high performance constraints across remarkably diverse architectures. Device advances have shaped the system-on-a-chip (SoC) landscape, including field-programmable gate arrays (FPGAs), hybrid FGPAs, full custom integrated chips, and commercial fixed-architecture platforms. The common thread across these platforms is the integration of complex distributed memory architectures, including caching hierarchies and local scratchpad memories (SPMs); in fact, the majority of silicon real estate is dedicated to memory and caching resources. Conventional automated approaches focus on functional components, specifically operations and arithmetic resource binding. Management of state and caching resources, however, are often treated as a secondary effect. In this work, clear representation and management of global state ensures that operational units are adequately used, and becomes a primary goal of the synthesis strategy. The importance of this effort grows as application memory demands scale beyond the capacity of local resources. Making efficient use of available data resources is an enormous ongoing challenge, and calls for a fundamental rethinking of the computational and memory models.

This research introduces a novel specification model with unique underlying data and execution semantics designed to enable cosynthesis across varying computational domains without sacrificing performance. Its focus is on application specification, restricting designer behavior while encouraging explicit and clear expression of designer intent. At its core is the hierarchical transactional model, a

formal, abstract model of computation that captures state and execution common across heterogeneous semantic and execution models. The model enables structured cosynthesis that is resilient to shifting platforms and varying constraints, all the while leveraging a unique data/execution encapsulation framework to guarantee scalable analysis. At the front end is a practical, understandable specification language that encourages concise design of complex applications.

1.1 From Codesign to Cosynthesis

The impetus behind software/hardware co-design is simple: software models are sufficiently abstracted from hardware, allowing easy adaptation to new architectures. A stringent set of abstractions shield software from changes in e.g. caching layout, protocol interfacing, and physical memory constraints, none of which affect correct execution. Comprehensive tool suites and well-honed design practices allow software to be practically scalable and are key to faster time-to-market.

In addition to easy adaptation, co-design is a mature field with a wealth of research, industry support, and tools to ameliorate challenges that may arise during software development. Device vendors maintain stable ABIs to assist development on shifting platforms, encouraging incremental design coinciding with platform upgrades. Co-simulation packages simplify validation and integrate well with existing engineering methodologies, facilitating short design iterations and quick exposure of otherwise-evasive bugs.

When software can no longer meet application performance demands (be it high throughput, low latency, and/or meeting real-time deadlines), designers employ hardware accelerators. Manually converting software into hardware is practical for small-to-medium applications and often involves significant algorithm restructuring to better leverage available concurrency. Mixing concurrent software and hardware, however, can quickly lead to a litany of component integration challenges. Reaching high performance, high utility designs involves significant engineering effort to navigate around concurrency bugs, resource arbitration complexity, and mixed protocol integration. At the same time, there are no global strategies for maximizing use of distributed, heterogeneous memories. The reality of state-of-the-art co-design exposes a gap between potential performance versus practicable design methods. Automated cosynthesis

1.1.1 The Case for Cosynthesis

In the most general sense, cosynthesis optimizes applications across varying computational domains. The most common instance refers to embedded system synthesis: the automatic creation of hardware accelerators interacting with a larger software application. At the very minimum, it includes scheduling and binding of bus interfaces and memories, along with synchronization infrastructure required to make it work coherently.

Specification-to-silicon cosynthesis holds tremendous potential to provide a fundamentally robust design methodology that not only adapts to shifting platforms, but exploits available resources to reach unprecedented performance. Its benefits are across the board: improved time-to-market, rapid development itera-

tions to determine system costs early in the design process, and effective architectural level optimization. Such a framework would allow designers to exploit novel memory architectures and complex intrachip networks, all the while equipping them with the tools they need to validate and verify otherwise difficult-to-debug concurrent applications.

1.1.2 The State of the Art

In reality, the heterogeneous computing landscape is rife with scaling challenges, ill-suited specifications that are incompatible with the underlying hardware, concurrency bugs, and poorly modeled system costs. Available cosynthesis tools simply cannot cope with meeting difficult constraints against such a diverse design space, especially as applications scale up in size. One of the fundamental issues is a distinct lack of a unified model capable of bridging different computational domains together. Without a comprehensive representation model, there is strong reliance on graph-based approaches and software-to-RTL models. Scalable, practical synthesis is possible with an expressive specification language designed around system-level optimization, rather than adapted from existing approaches. A unified computation model can accommodate vastly different data and execution models to exploit varying degrees of potential parallelism. It requires capture of dominant system costs and a high degree of flexibility in realizing behaviors across different classes of components.

1.1.2.1 Exposing and Modeling Dominant System Costs

When profiling applications, designers rely heavily on experience, as there are no straightforward ways to intuit a system's cost without comprehensive simulation and implementation. Even with rich modeling tools, the true costs of a system may not become apparent until the application is executing in silicon.

In current generation heterogeneous computers, application performance is dominated by the motion and storage of information. Computational pipelines vastly outpace memory interface speeds, often by several orders of magnitude. Hierarchical transactions were designed to expose and codify real system costs, particularly with respect to data motion. They allow direct capture and management of data motion costs to reflect modern architectural realities.

1.1.2.2 A Symbiotic Development Relationship

Robust design methods are clearly advantageous to designers. There is another key benefit: it has the potential to liberate commercial vendors and IC manufacturers from legacy interfaces. When a company releases an SoC product, they are beholden to uphold the same or compatible ABI for further versions of their product. The first iteration effectively sets in stone core architectural features that define its platform. Pressure to preserve these configurations comes from both sides: significant effort goes into developing compilers, simulators, libraries, etc. and vendors are reticent to redesign their tool chain. On the other side, customers demand backwards compatibility to avoid the risky task of reimplementing their application.

For example, a first iteration SoC may contain a centralized, cache-coherent RAM. In the next design iteration, newer research or device technology could make available e.g. distributed, lower power memories through crossbars. However, neither side is incentivized to adopt the technology because of rigidly coupled specifications paired with long, cumbersome design and verification times. Worse, customers' applications are tightly coupled to the platform's original architecture; leveraging the next generation's architecture may require a rewrite.

1.2 Contributions

The hierarchical transaction computational model lays groundwork for a unified semantic data and execution model. It codifies high-level application behavior through a carefully constructed semantic and syntactic model designed to ensure robust design methodology. Transactional data semantics guarantee verifiable behavior of latency-tolerant concurrent tasks. The notion of computational and state mobility supports maximal exploitation of parallelism – at their core, hierarchical transactions localize all state and control. This encapsulation enables use of hierarchy that is scalable to very large designs. As a result, access to the optimization space is not hindered by complexity in analyzing a specification.

At the front end is a novel specification language to directly represent hierarchical transactions. It focuses designer intent through specific limitations in how state is accessed. Designers are empowered with tools to directly identify and address concurrency bugs. The language is structured with synthesis in mind – it provides methods of expression that are confined to actions commensurate with

optimization. Designers express families of valid executions in a minimal format through high-level dependencies, type systems, and computational relationships. Low-level resources such as computational pipelines, protocols, and memories appear as interchangeable resources.

A corresponding compiler enables rapid behavioral simulations including fast identification of race conditions while providing direct paths to high-utility synthesis. Through a unique iterator specification, a compiler generates bounded-lifetime dataflow representations well suited for high performance computation. Often, in an embedded system, heterogeneity arises from the use of highly concurrent arithmetic pipelines. The primary bottleneck becomes the management and allocation of local state. This is addressed with a series of clustering heuristics that exploit controlled variable lifetime and high-flexibility transactional tasks to make efficient use of distributed memories and computational pipelines.

1.2.1 Computational and State Mobility

Traditional scheduling contains a notion of mobility for a given node/task – this describes its relative ability to "shift" in time, essentially describing its scheduling flexibility. Hierarchical transactions enforce a notion of computational and state mobility. This extends the concept of operational mobility to include a task's overall behavior, its intermediate and long-term storage, and its timing characteristics, especially across varying memory granularities.

There are three dimensions to the mobility of a behavior: (1) computational mobility describes flexibility of *the domain* that can implement the behavior; (2) state mobility describes *where* it may be realized, essentially representing how

tightly coupled the operations are to their associated storage; and (3) scheduling mobility describes *when* a task may begin or end. The tighter any of the three properties are constrained, the less opportunity there is to reach a feasible architectural implementation. Contrawise, greater flexibility across all three enables a design space amenable to architectural, system-wide optimization.

Computational mobility can only occur if the application specification is sufficiently abstract, e.g. it is not bound to any particular computation unit. State mobility requires that any computation be localized to its state – if a task needs to be done, both its state *and* its computation/control is treated as a discrete, mobile object. Scheduling mobility requires true timing decoupling through intrinsically latency tolerant design (intrinsically, as opposed to explicit, requires a fundamental guarantee *a priori* and a natively latency tolerant input specification).

Hierarchical transactions enforce true computational & state mobility through hierarchical encapsulation of state and control. All of the information a given behavior needs is encapsulated into a transaction, and its read/write interactions occur on well-defined transaction boundaries.

1.3 Overview

This research lays groundwork for a novel specification approach, evaluating its potential as a complete cosynthesis solution. As such, it is not a comprehensive exploration of the model. Instead, this dissertation details studies that exercise different aspects of the model with four goals in mind: (1) encouraging clear, concise application specification that communicates designer intent, (2) providing prac-

tical design tools to address correctness, (3) representing real control-dominated applications, and (4) enabling automated paths to efficient high-performance realizations.

First, Chapter 2 briefly surveys specification and representation models in cosynthesis. Chapter 3 introduces hierarchical transactions, including the novel concept of state and control mobility through hierarchical encapsulation.

To support clear and concise expression of the semantic model, a novel guarded rule-based language is presented in 4. It efficiently captures complex control as well as arithmetic-heavy applications. A full featured compiler and simulator provide front-end features to guide designs and discover correctness errors. Chapter 6 presents design analysis methods that demonstrate feasibly scalable isolation of race conditions. Additionally, algorithms are provided that directly translate selective portions of an application into control/data-flow graphs, outlining how hierarchical transactions can maps into different existing semantics.

Real-world heterogeneous systems exhibit a mix of complex control and high-performance computation demands. The final two goals represent either end of this spectrum. Chapter 7 discusses implementation of a microprocessor, describing control specification as well as practical race condition detection. Finally, Chapter 9 presents a high-performance cosynthesis methodology targeting hardware acceleration of Fast Fourier Transform (FFT), matrix multiplication, and image convolution. The presented near-linear clustering and scheduling heuristic scales up to million-node problem sizes.

Chapter 2

State of the Field

Cosynthesis (also called embedded system level [ESL] synthesis) is part of a broader co-design research field including hardware/software partitioning, software-to-RTL, and co-simulation. Each of these sub-fields have developed a diverse range of heuristic and exact solutions, many of which are surveyed in [86, 23, 33, 88, 87, 64, 71, 80, 84, 3]. Approaches range over a variety of language, semantic model, and domain & application-specific solutions. It is important to note that there is no single definition of cosynthesis – instead, automation is used to varying degrees depending on the problem setup, the target environment, and the assumptions underlying design flow. For instance, some approaches view cosynthesis as an extension to cosimulation: models are generated at a high level, then manually partitioned and synthesized into their respective domains. Others take a single-specification format and perform full automation down to silicon.

Despite a wide gamut of approaches, a unified solution has not yet emerged due simply to the sheer extent of the challenge – instead, a variety of methods have evolved to address specific architecture classes and problem scales. Unlike RTL, which has a relatively mature Boolean logic-based foundation, there are

as many variations of the problem setup as there are solutions. This not only complicates methodology comparison, but there are no straightforward ways of comparing generated design performance. Common benchmarks (such as JPEG and MPEG encoding/decoding) provide a similar basis for initial specification; however, divergence in optimization criteria renders any comparison uneven. The broad scope has prompted taxonomy classification suggestions [80]. The magnitude and breadth of computation, data management, and myriad interfaces mean that the heterogeneous design space is difficult to automatically traverse and particularly prone to scaling difficulties.

This chapter will discuss the common used models, particularly input specification format/underlying semantic models, and how that affects automation methodology. It will extrapolate fundamental assumptions that comprise both problem setup and results. The primary differentiators in the proposed hierarchical transactional model are: built-in scalability, data/control mobility in the input specification, and direct capture of memory capacity. These factors will be the basis for comparison against existing approaches.

This chapter first provides a general discussion on existing system representation formats (particularly graphs), introduces challenges arising from addressing/indexing, and is followed by an overview of existing work in the above subfields.

2.1 Behavior and State Representation Models

The single most common method of representing heterogeneous-targeted applications is the data-flow graph. Representation granularity ranges from fine grained operation-centric formats to coarse task-scale graphs aimed at abstracting lower-level detail. Hierarchy is occasionally used to target scalability[15, 26, 67, 75], often in the form of control hierarchy[56] (as opposed to state hierarchy). While control hierarchy simplifies reasoning of application flow, it does not ease handling of memory capacity constraints.

2.1.1 Early Computing Models

Early computing relied on an execution model based on the physical realities of its era. Both processors and volatile memories were of relatively similar speed. Large, nonvolatile storage were approximately an order of magnitude slower, but ultimately, arithmetic computing dominated costs in high performance applications. These constraints shaped the entire philosophy behind computing design, particularly in software and at the hardware/software boundary.

Much of the early technology constraints are still visible today in co-design/co-synthesis. The idea that hardware design centers around operations is a pervasive one, and has driven the field of high level synthesis towards operation-centric optimizations. As transistor technology has improved (through miniaturization), the performance gap between arithmetic operators and memory access speeds has grown tremendously. A significant portion of architectural engineering revolves around mitigating the gap through caching strategies, prediction/look-ahead, la-

tency hiding paired with deep pipelines, deep memory hierarchies, and a variety of manual memory localization tricks (i.e. design philosophies that attempt to keep operands as physically close to their arithmetic logic).

As a result, dominant costs in high performance software is the motion of data. General-purpose graphics processor (GPGPU) pipelines are a prime example: they accelerate high operational computations, but are largely useful only when computational requirements far exceed data requirements. A significant portion of accelerated applications runtime is dedicated to moving information to and from the memory attached to the GPGPU – in fact, standard design practices are centered around overlapping this transfer with computations reach maximum throughput.

2.1.2 Moving Towards Novel Models

Data-flow graph techniques straddle the technological boundary between operation-dominant and data motion-dominant systems. Graphs are a straightforward representation of operations, although they do not explicitly represent state – instead information is captured in the dependencies between operations (i.e. graph edges). As dominant system costs shift, so should representation models. An overview of data-flow graphs follow, along with a discussion on why additional control complicates verification in the codesign space.

2.1.3 Data-flow and Task Graph

A data-flow graph is a directed graph consisting of vertices/nodes representing behavior, often arithmetic operations. Directed edges represent information passing from one node to another, and therefore induce a dependency between the nodes. Graphs comprise the majority of existing research and commercial tools as they are easy to reason about and lend themselves to many well-understood heuristics. Adapting meta-heuristics such as simulated annealing and genetic algorithms is often straightforward[84]. Further, graphs can be directly scheduled[4, 16, 44, 91], as they contain the minimum dependency information required to apply any of the well-known scheduling algorithms.

Data motion is along the edges, between operations. Initially, edges are not bound to any specific resource element; instead, a cosynthesis solution maps these edges to either software registers, hardware latches, and/or bus transfers. When partitioning an application into hardware and software domains, data motion costs are measured by observing the amount of information that flows across a partition's boundary – this eventually becomes the hardware/software communication.

Data-flow is designed to model a specific class of application and thus avoids the issue of arbitration. For control/data-flow graphs, if they *do* include arbitration nodes, formally verifying correct behavior becomes very difficult – deadlock is common when concurrent paths arbitrate resources over varying latencies[18, 61, 60, 11, 24]. When multiple elements need to share a single resource, arbitration becomes the synthesis tool's responsibility, inducing performance-limiting synchronization mechanisms. When these mechanisms show up in the cost model,

they effectively pre-partition the graph (since the alternative mapping bears a prohibitively high cost).

Hierarchical transactions, on the other hand, are an abstraction layer above data-flow. They capture more information and specify very large families of potential mappings and executions. Interaction between shared state resources are explicit – there are no implicit arbitration points. Instead, any potential reordering of shared data writes are passed to the designer to resolve, thus avoiding race condition situations. Better yet, there are direct paths to data-flow graphs. A method in chapter 9 selects one of the families of valid execution and localizes data along the edges. In situations where data-flow may not be appropriate, the synthesis tool may leverage existing arbitration techniques to ensure correct behavior. For example, control-heavy sections of an application well-suited for software may use built-in atomic instructions to arbitrate shared data.

2.1.4 Latency Tolerance

One of the aims of our model is to leverage existing research that is compatible with presented semantics. Domain-specific models can be appropriately targeted depending on system constraints to make maximal use of available platforms. Therefore, transactions leverage existing latency tolerance technology [13, 21, 22, 49] to manage control, accurately reflect the way heterogeneous components interact, and apply existing verification techniques to ease designer burden.

2.1.5 Transactions, TLM, and Transactors

2.1.5.1 Transaction Level Modeling

Transaction Level Modeling (TLM) is a popular method of improving simulation speed, and therefore improving design quality and time-to-market. The primary "transaction" in TLM is largely focused on simplified data motion isolated from bus semantics. Rather than simulating complex concurrent state machines, they simulate black boxes emulating those machines with dramatically lower computational requirements. TLM was introduced as a way of performing sensible design exploration and validation of complex systems [12]. Recent research has emerged in directly synthesizing TLM designs, particularly individual parts of the system (e.g. bus communication [72], emulation/verification/simulation [62], and task-centric solutions[57]).

2.1.5.2 Synthesizable TLM

Recent pragmatic system synthesis approaches transform SystemC TLM 2.0 models into Kahn Process Networks (KPNs), assuming the inputs follow certain bounded looping behavior[38, 65, 53]. Since the approach begins with a C-based specification, state and control must be manually preallocated into smaller subtasks to fit into their target model (task partitioning). As a method of codifying interactions, this approach has proved capable in full-stack specification to silicon, in large part due to guarantees in the KPN model. Unfortunately, not all applications in this space can be represented with KPN, and thus the application

space is limited. Further, there were few scalability guarantees discussed in their respect papers.

2.1.5.3 Nested Transactions and Software Transactional Memories

While the hierarchical transactional model differs in critical ways from current TLM work, it does share semantics with nested transactions applied to transactional software memories and database systems. These models have proved to be sound and complete, wrapping data and control [2, 1]. The same challenges facing database coherency appear in heterogeneous systems, for many of the same reasons – primarily, databases are distributed to share a large workload over multiple computational units. Since these units (database server nodes) each run at their own pace, they cannot be statically scheduled. All of the information must remain coherent over a complex data network. Similarly, cosynthesis must ensure correct data exchange over buses (rather than ethernet) using arithmetic accelerators (rather than database nodes). Transactional memories [37, 50, 43] are similar models addressing the issue of data coherence.

This work differs in that it includes functionality in the transactions rather than just state. Further, latency tolerance is explicitly specified via a rich token-based abstraction, thus concisely describing functional dependence. While the problem setup may share similarities, the heterogeneous design space exhibits dramatically different costs and must be tuned accordingly.

2.1.5.4 Transactors

Asasnovic's work in [7] transactors is another closely related semantic; however, in their approach, communication is handled through explicit FIFO channels between transactors, and follows a BlueSpec-like model. Further, they did not exploit hierarchy. Work by Belarin [9] uses the transactor nomenclature, but focuses on bridging high-level and low-level models, eventually using Hierarchical Finite State Machines (HFSMs) for code generation. The latter model puts no bounds on control, and can run into many of the same problems that traditional control flow graphs suffer.

2.1.6 Hardware/software Partitioning

Efficient hardware/software partitioning is a well-researched topic[86, 83, 30, 51, 41, 42, 40, 52], leveraging popular heuristics [8] including simulated annealing (SA)[30], genetic/evolutionary algorithms (GA)[76, 6, 28, 84, 45, 75], dynamic [60, 66] and linear programming[68, 6]. The myriad approaches are surveyed in [84].

Partitioning is generally grouped under the co-design umbrella and involves deciding which functional elements from an input specification should be implemented on software versus hardware targets. Input specifications are relatively abstract functional descriptions in order to facilitate this flexibility. Heterogeneous architectural targets are either assumed to be capable of implementing all functions, or are categorized into groups.

Input specifications are almost universally represented in the form of control/-data graphs. The solutions assume a one-to-one mapping between input graph/-tasks and the target architecture, or they construct mapping data structures to represent all valid architectural maps. The latter can grow exponentially. As a result, they are not amenable to large designs that exceed locally available hardware resources.

2.1.7 An Overview of Cosynthesis Models

Early work by [34, 35, 89, 17, 79] established the basis for future cosynthesis work. These approaches universally use tasks graphs (with some applying hierarchy for control compactness[15]). Variations include timing annotated and periodic task graphs [28, 25]. Multi-mode approaches [55, 70] focus on mapping task graph sets according to different modes of operation.

The Hierarchical extended Task Graph (HeTG) introduced in [59] extends basic task graphs with control nodes, specifically fork, join, and select nodes. Their primary goal is to allow powerful control that is compatible with scheduling. Further, they encapsulate behavior into hierarchical nodes, allowing coalescing of control logic (i.e. a hierarchical task is responsible for the control overhead of its children). This work applies similar techniques, but pushes data encapsulation to the forefront.

Chinook [17] accepts an HDL-like input specification, adopting RTL-style data and control semantics; that is, data and control are not constrained and therefore the design do not express state & control mobility.

The Daedalus system [82, 69] accepts KPN-based inputs to perform full system synthesis, taking advantage of accurate system modeling to correctly model and optimize the system accordingly. They benefit from KPN's guaranteed boundedness and straightforward data exchange; however, not all applications are well-suited for KPN, as it precludes any application that requires control.

Hierarchy has been used to different effect in cosynthesis models. For instance, [39] leverages hierarchy to describe alternative realizations of the same coarse-grained task – this simplifies the search space and improves convergence.

The PeaCE [36] multimedia research included full-stack cosynthesis based on the Ptolemy[63] mixed computation modeling system. It restricts Ptolemy to two models: a "piggybacked" synchronous data flow (SDF) and an extended "flexible" finite state machine (fFSM)[56]. Piggybacked SDF models are introduce control tokens, much in the same way [59] extended task graphs. Without formal deadlock resolution, traversing this design space can be difficult and cumbersome. The fFSM is particularly interesting because it constrains state machine transitions along hierarchical boundaries, thus confining control scope; this goal is shared by hierarchical transactions and motivates the encapsulated token model.

2.1.7.1 Magellan

The MAGELLAN [15] project shares many philosophical goals as this research, and provides a complete cosynthesis package. At its core, it iteratively maps hierarchical control/data-flow graphs onto a heterogeneous architecture, decoupling specification from target. Its input semantic uses hierarchy primarily to coalesce control complexity, thus avoiding the need to identify control symmetry. It fur-

ther leverages hierarchy to make multi-grained moves during optimization. The separate graphs – application and architecture – are constructed into a new graph representing groups of possible architectural maps. A heuristic iterates over this structure and incrementally improves latency and area.

The demonstration JPEG application is relatively small, although a large part of their compaction arises from hierarchy. The presented data is notably fast (on the order of 11 seconds to optimize the design) – however, there is no discussion on how the approach scales. For a design on the order of tens or hundreds of thousands of nodes, it is clear that the architectural mapping alone would render the approach infeasible. Hierarchy is an option to keep this in check, but it is not clear that a large, deep hierarchy will converge. Their optimization heuristic requires the ability to make multiple, multi-scale moves; if each step is too computationally or space intensive, then the heuristic simply would not get the opportunity to try many system configurations.

2.1.7.2 Metropolis

Metropolis[10] similarly shares the same goals as this research. Their approach is built on concurrent communicating processes (CCP)[46], a well-researched, clearly defined formal model for concurrent tasks. CCP evolved from message passing where deadlock/liveness issues were common[5] – thus, a heterogeneous application model based on CCP benefits from this research. Metropolis also represents an abstract model (philosophically similar to hierarchical transactions) and is clearly capable of representing lower level architectural models. While CCP is an abstraction well suited for representing complex control and semantically sim-

ilar applications, it does not directly model data/intermediate operand storage. As such, its data modeling is limited to the information captured through message passing interfaces. In situations where the task contains persistent state, its interactions are treated as abstract behaviors rather than explicitly codified state modification.

Its quasi-scheduling technique is amenable to hierarchical transactions through two paths: first, by converting transactions into Petri nets (similarly to the method used for data-flow projection), Metropolis' methods can be applied directly. When constructing this Petri net, Metropolis effectively selects a specific execution path through its internal mapping network before passing this to the scheduler to determine timing. An alternative transaction strategy can select one of the valid families of execution and apply the techniques without the Petri net intermediary. The primary benefit would be leveraging Metropolis' communication overhead optimization.

2.1.8 Bluespec

Bluespec is an operation-centric rule-based guarded action language. It rides on a FIFO-based communication network that is ultimately realized through handshaking (ready and stall signals). Its power is in its ability to meld high-level functional type semantics with low-level RTL-style semantics. Designers reach fast design convergence, quickly determining the true costs of the system. The methodology is compatible with RTL design, and thus makes the transition smoother.

While low-level signaling and clock-scale design are easier to use, they also leave the Bluespec compiler with few options when realizing the design. Scheduling is relegated to the limited freedom the designer allows (if they, for instance, explicitly design with several clock cycles of latency tolerance in mind).

Despite being natively latency tolerant, the compiler ultimately schedules the design by packing as many rules into a single cycle as it can. It is the designer's responsibility to ensure timing closure by running synthesis and then manually cross correlating the violating paths to the original specification. They insert pragma statements manually rescheduling rules causing timing violations. The Bluespec compiler makes it remarkably simple to institute necessary scheduling and resource allocation changes due to its robust type and module instantiation/-parameterization system. The underlying FIFO model allows a certain degree of separation from clock timing where hard bounds are unneeded.

Unless the application is very carefully planned with the compiler's code generator in mind, the resulting RTL can contain tightly coupled components via its handshaking control signals. These wires scale poorly with the size of the application, making timing closure increasingly difficult with scale.

2.2 Software-to-RTL

The class of C-to-RTL/Software-to-RTL research and commercial products have reached maturity and (for applications that suit their assumptions) commercial viability. These tools directly address hardware acceleration of selected components from a sequential procedural application[47, 34, 27, 31, 19]. Geared

towards high-arithmetic algorithms, their primary task extracts maximal concurrency from a sequential specification through an assortment of advanced loop and pointer analysis heuristics[71, 14, 77]. They create data-flow graphs which are mapped into RTL components while their input/output interfaces are mapped onto buses.

2.2.1 Model Representation

As a result of adopting data-flow models, these approaches largely suffer from the same drawbacks that existing approaches have. While they do allow fast paths from software to hardware, ultimately, C is an ill-suited language for hardware[29].

IP Reuse IP reuse is touted as a driving force for using C-to-RTL. For applications already written and tested in C, the idea of simply applying a turn-key solution to existing code is immensely appealing. In the real world, this design flow does not pan. Fundamentally different assumptions in the sequential execution model yield diminishing returns in extracting concurrency from a sequential program[29].

Aside from the efficiency argument, C applications often require engineering effort to comply to the C-to-RTL tool's subset of C. RTL models are limited to the expression power of a finite state machine (FSM), and thus the tools can only accept applications that are isomorphic to FSMs. Further, almost every nontrivial C program uses dynamic heap and stack allocation; there are noteworthy heuristics that can aid this problem [85, 78] but must carefully avoid intractability. Thus, in the real world, existing IP must often be rewritten to allow the compiler

to generate static hardware. The code must be sanitized to remove architectural optimizations such as cache optimality.

2.3 Moving Towards Hierarchical Transactions

A significant amount of research has addressed co-design and cosynthesis through approaches. While graphs are an excellent method of reasoning about operations in high-arithmetic applications, they are ill-suited for resolving complex control or data arbitration. It has become evident that a new model needs to address this issue while still allowing high-computational portions of an application to leverage existing work.

Hierarchical transactions can be targeted towards multiple semantic models, thus granting the benefit of data coherence over large-scale data sharing protocols (similarly to database systems), and yielding high performance through graph-based approaches where appropriate (leveraging advances in modern cosynthesis). The model has been carefully designed to allow a broad spectrum of synthesis techniques that are suitably adaptable to the often incompatible semantic and execution models that underpin real-world heterogeneous systems.

Chapter 3

Hierarchical Transactions

The hierarchical transactional semantic data and execution model is a novel approach to specifying complex designs across the control and data spectrum. The model supports a concept of state and control encapsulation, enforcing local behavior throughout an application. Through careful design of its execution semantics, the model leads to design analysis and synthesis solutions that are both practicable and scalable. It brings state capture to the forefront, ensuring access to automated exploration of different memory management architectures.

3.1 Abstraction

A semantic data and execution model captures either abstract behavior or reflects a specific architecture. The advantage to the former is a clear delineation between high-level application behavior and its target platform. The latter exposes important cost metrics critical to efficient optimization. Historically, common practical models have attempted to balance both model classes. In RTL, the execution model is inherently concurrent and can become unintuitive as an

application grows; complexity is typically managed through finite state machines that abstract complex concurrent interactions and enable sequenced behavior and easy-to-understand semantics. On the other side, the C programming language maintains a primarily architecture-agnostic specification, but contains inline assembly, pointer arithmetic, and e.g. volatile type classification to reach low-level architectural optimizations.

Hierarchical transactions (HT) are a purely abstract model – however, what sets them apart is they are designed from the ground-up to adapt to directly modeled architectures commonly found in the heterogeneous computing space. More importantly, HT enables better use of existing semantic and data models through computational and state mobility – it specifies a larger class of configurations that provide multiple mapping paths to existing hardware or software models. For example, there are a variety of options for constructing a synthesis path from HT and RTL, depending on the particular state and computational constraints for the target architecture.

3.2 Challenges in Cosynthesis

Hierarchical transactions directly address cosynthesis models through a combination of features that unify heterogeneous models into a codified, analyzable, and scalable manifold.

Concurrency One of the primary benefits for shifting computation into hardware (besides potential power savings) is its tremendous concurrency/parallelism

potential. To reach this potential, the model natively supports and encourages heavily concurrent behavior. It provides direct paths for mapping RTL, DSPs, and GPUs (all of which enable high degrees of concurrency), all the while ameliorating common concurrency problems through a race condition/deadlock reporting mechanism.

Latency Tolerant Software, FPGA RTL, DSPs, and GPUs all exhibit vastly different timing characteristics. To enable synthesis across these domains, all of the specified behavior in an application is required to be latency tolerant. Traditionally, dynamic timing flexibility had the drawback of increased overheads and performance penalties. However, this model is designed around the ability to optimize unnecessary dynamic latency tolerance in favor of static schedules based on the architectural optimization criteria. In essence, the compiler decides where to preserve dynamic latency tolerance based on its assessment of implementation costs. Latency minimization, therefore, is a synthesis constraint. From the designer's point of view, they need only specify correct gross ordering, rather than hard timing (which is relegated to resource models where hard timing is critical, e.g. bus or protocol timing).

Transactional The combination of concurrency and latency tolerance are key to model robustness. However, they also lead to difficult-to-find concurrency bugs, making verification cumbersome and time-consuming. The solution is to enforce all data interaction to be well-defined and analyzable along transactional events.

If the tool knows how and where data and control information are accessed, it is able to identify potential concurrency bugs very early in the design stage.

Copying Semantic A copying semantic gives each task the illusion that it has its own local copy of all data & relevant control state. Decoupling dependent tasks means synthesis tools can realize any part of the behavior in any domain. Encapsulated behavior further alleviates difficulties in implementing cross-domain control. It is an optimization opportunity that maximizes architectural search space: the synthesizer has full knowledge of data and control variable lifetime, and each transaction copy is a potential design decision that can easily be optimized out where unneeded.

Hierarchical Another consequence to concurrent, latency tolerant behavior is potential exponential growth when analyzing execution paths. In order to determine data operations that may lead to concurrency bugs, every possible path must be explored. Hierarchical design analysis alleviates this, especially with the combination of transactional data interaction and a copying semantic. Every subtask or group of transactions can be completely isolated from the rest of the design, dramatically limiting the number of exponential decision paths for well-partitioned input specifications. Pseudo-linear analysis is now possible since high-order polynomial or exponential growth on the local child network is eclipsed by the linear traversal of the hierarchy – i.e. 2^n is insignificant for small n. Further, this type of hierarchy is amenable to optimization during synthesis: the overhead traditionally

associated with hierarchy is optimized away since that overhead is codified in the copying semantic.

Controlled Array Access Most, if not all, high-computational applications (e.g. signal processing) require the ability to traverse (multidimensional) arrays. Traditional specification languages provide arbitrary indexing into arrays which has historically made concurrency extraction very difficult, forcing tools to rely on heuristics and designer feedback to control array stride. Indexing also complicates accurate determination of variable lifetime. Hierarchical transactions provide an alternative to arbitrary indexing. Repetitive/looping array behavior is specified through well-defined, analyzable iterators to achieve the same specification flexibility as traditional looped code. The approach allows large arrays to coexist with the copying semantic – only the current iteration needs to be copied locally, and iterators grant information to the compilers about precisely which subsets will be copied.

From a specification clarity point of view, iterators express designer intent more concisely and accurately than arbitrary index functions. They signal to the synthesis tool precisely how data in the array is to be accessed.

3.3 Semantic Model

3.3.1 Hierarchical Transactions

The core component of the semantic model is the hierarchical transaction.

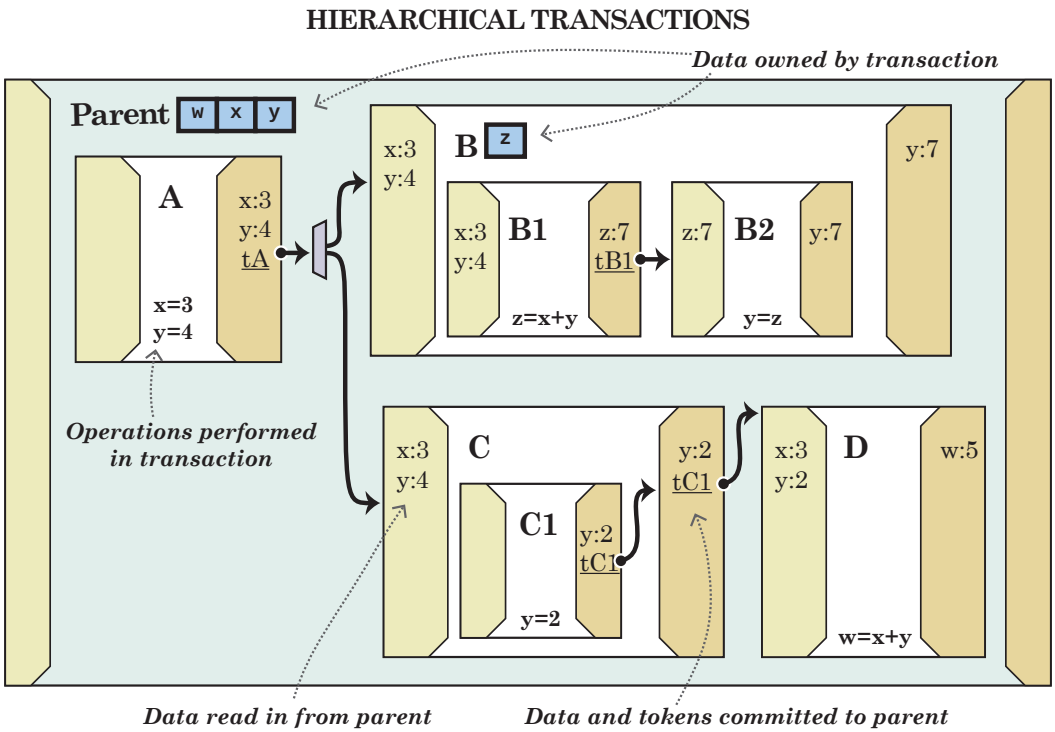


Figure 3.1: A small hierarchical transaction design, annotated with possible variable values from a single execution. Each transaction shows variables that are read-in on the left, and variables and tokens committed on the right. The PARENT transaction owns variables w, x, and y. When each of its child transactions begins, they make a local copy of these variables. Transaction B owns a variable called z. Tokens are shown in italicized underlined text in the commit lists. Transaction C illustrates hierarchical token passing – tokens may only pass through their parent to reach higher transactions, following the same commit rules as data. Transaction C1 creates a token *tC1* which is passed to C before finally being passed to D.

Definition 3.1. An *action* is a set of one or more statements. Each statement may be an arithmetic operation, an assignment (state updating), conditional statements based on the local state, or token creation. Every action in the transaction executes atomically and concurrently. All reads and conditions are evaluated first, then all of the writes are performed independent of ordering.

Definition 3.2. A *transaction* consists of: a *guard*, local state, and either an action or an acyclic dependency graph of child transactions.

3.3.2 Overview

Every behavior in an application is described by transactions exhibiting well-defined start and end events. Data and control scope is strictly local – i.e. outside transactions may not modify or observe internal variables or control state. Transactions may contain child transactions arranged into a directed acyclic dependency graph, thus forming a hierarchy of transactions.

All transactions encapsulate behavior, data, and control within a strict local scope. When a child transaction begins, all of the data that it needs is copied from its parent. When it completes, the transaction may commit changes to its parent. Every transaction (subtree) can be viewed as a pure function, accepting input data from its parent and returning updated values. All of their data (owned and inherited) is local to its scope. This interaction occurs strictly between child and parent only – a grandchild cannot directly access its grandparent's data; it must go through the intermediate transaction. It is important to note that the final implementation does not need to maintain this copying. Such a system

would incur large overhead. Instead, binding and scheduling determines low-level implementation while preserving the application's behavior.

An example design is shown in fig. 3.1. The figure illustrates both hierarchical encapsulation, data interactions on transaction boundaries, and hierarchical control token passing (described below). Both transactions B and C get their own copies of variables x and y. B1, the second child of B, reads the local value of y from its parent. Transaction C1, meanwhile, commits a new value of y to C before it finally gets committed to the parent.

3.3.3 Computational and State Mobility

The most significant element of hierarchical transactions is computational and state mobility. Every transaction strictly encapsulates all of its internal behavior through an abstract copying semantic that guarantees state is always local. These attributes ensure a high degree of mobility and flexibility when determining where to map architecturally. For instance, for a given transaction, if its parent exists in software while it is mapped onto a hardware accelerator, the copying mechanism becomes a bus transaction. If the same transaction remains in software, but is targeted towards a multicore implementation, then the copying manifests in a partitioning of the state. Rather than manage memory coherence, the synthesis tool may map the local state to a separate memory segment and allow it to run concurrently. If the transaction remains in the same thread as its parent or peers, then the synthesis tool may remove copying altogether and ensure the same behavior by scheduling transactions in a particular order (essentially sequentializing

them such that the data dependencies preserve the illusion of copying without the unnecessary overhead).

3.3.4 Token-based Control

Control flow between transactions is handled through token-passing – when a transaction completes, it creates tokens that determine the next set of transactions to start. Each transaction contains a guard, describing the tokens that must exist before it can begin. Tokens are identified with a unique name. Each transaction has a *guard* expressed as a Boolean expression of token identifiers, enabling complex token control. Once the guard has been satisfied (i.e. the required tokens have been created by other transactions) all of the tokens involved are consumed atomically. Thus, a transaction is *guarded* by tokens. Tokens primarily describe functional dependencies between transactions (independent of timing). Their existence implies a local dependency network similar to traditional control-flow techniques.

Limitations in control expressability ensure that control lifetime is always bounded. The semantic model's policies disallow transactions from arbitrarily observing tokens. A token's lifetime begins when its creating transaction commits, and it ends once it has been consumed. This decision enforces a limit on the lifetime of control state. To cover many practical applications, conditional token creation is allowed – that is, tokens can depend on the value of a variable. In general, data-conditional control is difficult to bound both in lifetime & scope – verification requires knowledge of the temporal longevity of any control decision, otherwise it leads to the halting problem.

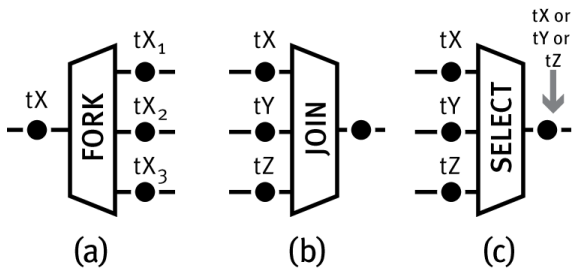


Figure 3.2: (a) Fork nodes replicate tokens when multiple transactions are guarded by the same token. (b) Join nodes wait for all input tokens to arrive before generating its output token. (c) Select nodes wait for any input token, discarding the remainders.

In this model, however, every operation (including token creation and consumption) falls on a well-defined transaction event. The compiler can always fully analyze control dependencies, conditional or otherwise. Whole-system analysis for well-constructed hierarchies is pseudo-linear, since any high polynomial or exponential behavior for a small number of elements is insignificant. Thus, the compiler leverages abstraction to keep average runtime linear – it can very quickly determine all of the conditions that lead to a given transaction's creation. The tool always knowns the transaction boundary that a variable falls on and can determine whether a token condition is satisfied during its execution.

3.3.4.1 Combining and Forking Tokens

Latency tolerant designs require complex control mechanisms to implement real applications[21, 22]. Three control nodes manage merging and bifurcation: Fork, Join, and Select. If multiple transactions are guarded by a token, the

token gets replicated and each transaction gets its own copy. This induces a FORK node, where the created token forks into multiple transaction guards, as shown in fig. 3.2(a). Abstract synchronization is provided through a JOIN mechanism shown in fig. 3.2(b). A transaction guard of "tokenA and tokenB and tokenC" implies that the transaction will wait for all three to exist before it may fire. Thus a JOIN node is inferred and automatically inserted into the network at the appropriate location. Complex control applications often require an "either-or" selector. A transaction may specify a guard that states this relationship through a guard of e.g. "tokenA or tokenB or tokenC" shown in fig. 3.2(c). A generated SELECT node will wait for any of the tokens to arrive before proceeding. The subsequent tokens are ignored. SELECT nodes proceed regardless of the order in which their input tokens arrive.

3.3.5 Latency Tolerance

Latency tolerant design is a natural approach to heterogeneous design, particularly because behavior that is decoupled from timing encourages the design of a *family* of executions, rather than a fixed, implied schedule. Covering an array of valid execution sequences allows tools to evaluate implementations across multiple timescales, supporting true decoupling from application specification and improving design resilience against evolving architectures. The methodology forces designers to reduce their specification to a minimal description of the problem being solved. This frees them from including hard timing details at a large scale, which may not be relevant until design space exploration, when tools have a better sense of costs. Hard timing details generally affect lower-level implementation,

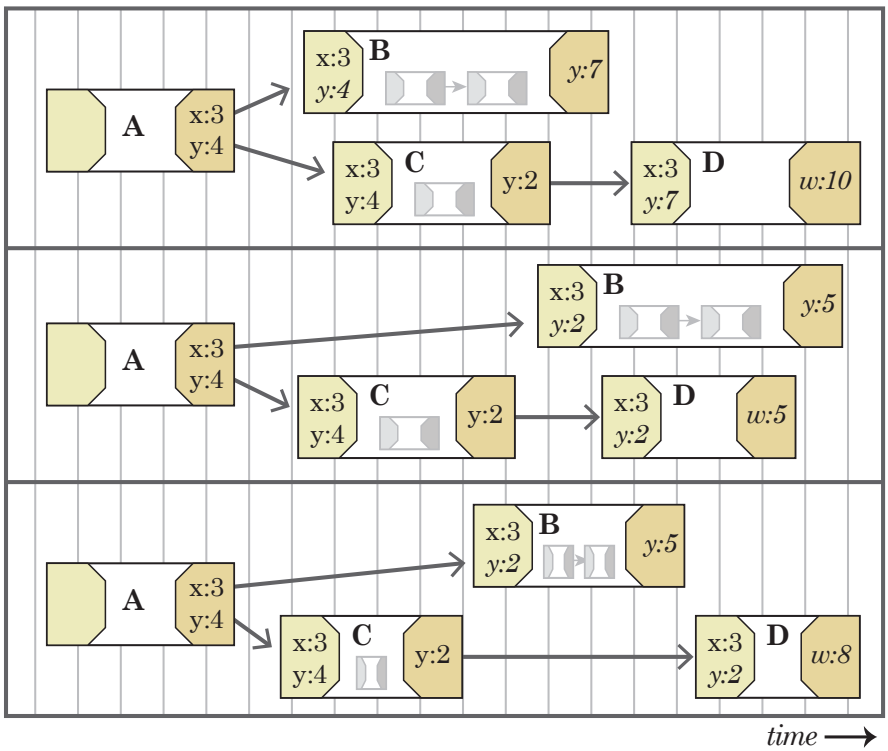


Figure 3.3: The same transactions from fig. 3.1 shown executing in different orders. As is demonstrated, the internal workings of transactions B & C do not need to be included. They are abstracted along the transaction boundaries. This idea is crucial to bounded concurrency analysis.

and these constraints are captured in the resources rather than the application specification.

Work in the area of latency tolerant design typically refers to gate-level isolation of clock domains. This semantic has generalized the approach, but applies similar control semantics through the token-based control network presented above [21, 22].

3.3.6 Upholding Abstraction

Hierarchical transactions uphold a combination of functional, temporal, and spatial abstraction:

1. Temporal abstraction is achieved through latency tolerance and by bounding data/control lifetime. Application specifications omit explicit timing information, reducing the problem to fundamental functional dependencies. Transactions tie data/control lifetimes to the length of the transaction.

2. Functional abstraction comes about by wrapping all behavior within finite-length transactions. The internal behavior of a transaction can be abstracted at the transaction's start and end events.

3. Spatial abstraction is enabled through locality of data/control. All design variables and control decisions are local to their transactions. Data containment means operations do not need to go beyond their local transaction scope.

3.3.7 Concurrency

To encourage maximal concurrency wherever possible, hierarchical transactions are designed to be concurrent unless otherwise specified. With the added parallelism of concurrency comes a host of difficult-to-find concurrency bugs. Typically, these problems emerge when concurrent tasks share resources, and the arbitration of those resources occurs in an unexpected order. Specifically, they give rise to race conditions (where data is written multiple times by concurrent tasks),

potentially leading to deadlock or starvation. Decades of research have provided designers with tools to debug these notoriously cumbersome bugs.

Concurrent transaction commits to the same variable have a high possibility for race conditions. Figure 3.3 illustrates how transactions may commit different values in different execution orders (using the same design from fig. 3.1). As shown, the values `y` and `w` are written with different values in each execution. When a designer specifies an application in this form, they are indicating to the tools that *this is correct behavior.* The designer is essentially letting the tool know that functional correctness does not depend on the value being overwritten by multiple transactions. In situations where this may be critical, the compiler quickly identifies race conditions (sec. 6.2.5), reporting them to the designer to resolve.

Chapter 4

Hierarchical Guarded Atomic Rule-based Language

4.1 Introduction

A fundamental rethinking of cosynthesis specification semantics provides the opportunity to explore a novel domain-specific specification language better suited to the task.

With any new semantic model comes the choice between designing a new specification language or writing a library over an existing language (leveraging existing technology to form the meta-language, i.e. compile-time language). The former requires a significant amount of initial work, especially in testing as language coverage is an often intractable problem. The latter library-based approach is cumbersome, making designs difficult to express while potentially limiting specification strength due to inherent drawbacks from the parent language. This research explores the former path via a language that directly and compactly reflects its underlying transactional semantic model. Despite significant setup effort, the

payoff is in the ability to concisely express complex designs without any potential pitfalls.

4.1.1 Existing Languages

There are many programming, hardware design, and abstract model languages in use; most reflect their respective tools' function. Often, however, languages are used simply because of familiarity. C is one such example.

C, however, is ultimately an implementation specification, not an algorithmic one. Real-world high performance C code is optimized for cached CPU architectures, and thus is built on certain assumptions about how information is exchanged in the system. Optimized C is structured around different system metrics, particularly in arithmetic versus data motion costs. For instance, since floating point operations are generally expensive in microprocessors, arithmetic-heavy C implementations will improve performance by introducing conditional cases when the value is zero, avoiding the unnecessary computation. In an ideal specification, however, such an optimization would not appear; it would instead eliminate any implementation-specific details in favor of pure algorithmic descriptions.

Bluespec[1] is a rapid development language that enables very fast design iterations of operation-centric applications. Quick design iterations allow developers to determine the primary costs of the system early in the design process – often, these costs can be counterintuitive or non-obvious. Encouraging a rapid design cycle

[1]BlueSpec® is a registered trademark of Bluespec, Inc. SystemVerilog® is a registered trademark of Accellera, Inc.

exposes the costs, and allows designers to manually restructure their architecture accordingly.

Traditional approaches to co-design use SystemC or similar languages. Recent work in Bluespec SystemVerilog[1] [58, 73] applies guarded rule-based reactive techniques to this field, and has demonstrated the trade-offs involved. In this work, we present a custom language aimed at directly capturing hierarchical transactional semantics, enforcing specification limitations as an efficient representation.

4.1.2 Aesthetics

Specification aesthetic refers to an application's ability to clearly and unambiguously express the designer's intent. In its purest form, a given specification would aim for true implementation independence boiled down to a concise, practical, and understandable description of its core algorithms.

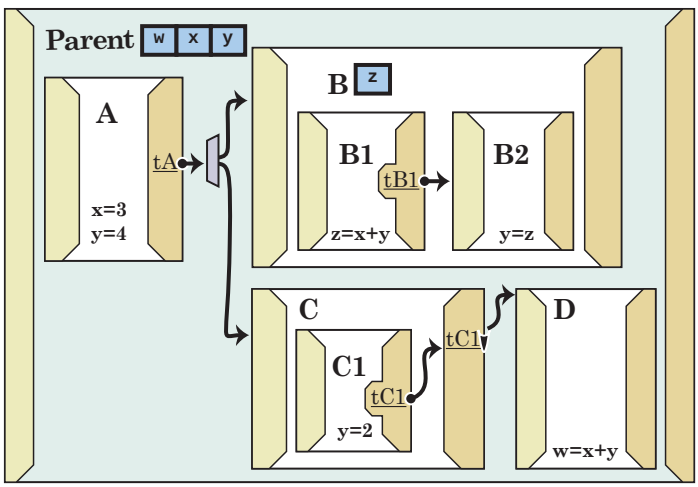
HTL syntax is modeled after Python for its attractive minimalist syntax that is well-suited for algorithm-level specification. Further, whitespace-defined scope provides a clean visualization of hierarchical scoping without the clutter of e.g. semicolons; at the same time, whitespace-tied blocks are difficult to manage for depths greater than 4 or 5, therefore encouraging designers to break up the design into much smaller pieces. This has the added advantage of encouraging more hierarchy than is strictly necessary, a property that grants analysis tools more flexibility. Fine-grained, deep hierarchies come at no additional cost. Unlike Python (or other hierarchically scoped software languages), there isn't a scope context change or stack overhead.

4.2 Core Language Philosophy

4.2.1 Concision, Meaning, and Abstraction

Conventionally, an application may be specified with a specific architecture in mind. As the architecture evolves, some of those elements may not be reasonable assumptions. Rather than rely on the the input specification to detail implementations suited for the architecture, it should describe the overarching goal with as little extraneous information as possible. This is one of the core issues plaguing C-to-RTL tools: C specifications for e.g. signal processing applications are not inherently concurrent, nor are they designed around the potential for distributed memories that may not be able to maintain coherency without significant overhead. There are more subtle effects as well – hand-optimized FFT implementations may include zero-checks to avoid expensive floating point computations. When translated to RTL, those checks introduce unnecessary performance-hindering synchronization. More significantly, since the original specification targets a single-access, large memory, it is the compiler's responsibility to infer local strides of data access which can be mapped into small, distributed memories.

The ideal cosynthesis language decouples algorithm specification from implementation, while still supporting synthesis of high-performance realizations. The language presented below aims to fulfill this goal.




```
rule Parent:
  var w,x,y:Bits(8)
  rule A:
    x=3
    y=4
    create tA
  rule B(tA):
    var z: Bits(8)
    rule B1:
      z=x+y
      create tB1
    rule B2(tB1):
      y=z
  rule C(tA):
    rule C1:
      y=2
      create tC1
  rule D(tC1):
    w=y+x
```

Figure 4.1: Example of token-guarded hierarchical rule language used to specify transactions. This code describes fig. 3.1.

4.3 Rule-based Language

The Hierarchical Transaction Language (HTL) allows direct specification of transactional semantics in a clear, concise way. The fundamental task expression is a guarded, hierarchical rule. Guarded rules provide a natural fit for reactive, concurrent systems, efficiently representing latency tolerant, high-level behavior. Every rule describes how a transaction begins along with its child relationships or its internal functionality. Fig. 4.1 outlines the syntax, providing code for the transactions shown in fig. 3.1. Indentation specifies scope, similar to the Python language. The language is statically typed and scoped, thus all variables must be declared and their types must be statically resolvable. Types are expressed using compile-time "duck typing" – as long as an object can implement the specified behavior, it is a valid type relation. The data types are constructed from components

in a library of generic type components commonly needed to express applications. In the example shown, the `Bits` class represents arbitrary finite-length integers supporting standard arithmetic operations, concatenation, and logic-level bitwise operations (shifting, masking, etc.). They translate to e.g. registers in hardware and character arrays in software.

Scope (namespace and variable access) is static, tied to the transaction generated by the rule.

4.3.1 Meta-language vs Language

There are two classes of specification: language features that directly describe the target semantic or execution model, and a meta-language that adds one layer of abstraction through compile-time expression. Put another way, a meta-language allows code that generates code. As a language's design evolves and the demands for concise expression of its behavior increase, a common lesson learned by language designers is that any practical specification language benefits tremendously from a meta-language. Automating and parameterizing repetitive behavior and variable types improves readability, and broadens the usability of a given specification. Without a meta-language, for example, a designer would need to manually specify separate design instantiations for different types or architectural choices, despite using the same fundamental algorithm.

Moving beyond trivially sized problems, it is beneficial to allow expression of meta-behavior; that is, language features that produce language that is then translated to compiler data structures and eventually code. A meta-language, in this context, refers to specification code that is used to describe design parameters,

i.e. behaviors that are indirectly related to the task that needs to be done. In the C++11 specification, for instance, template metaprogramming is a powerful technique that allows parametric classes with rather complex generating behavior. One of the drawbacks, however, is it is syntactically unwieldy and provides a different expression language over an existing one.

An incidental benefit of transactional semantics is rich static analysis. The same guarantees that allow data & control mobility also grant the compiler valuable information about the behavior of variables. If a variable is written once and read the remainder of the design, it will determine its write-once value and propagate it throughout the design. By adding this same functionality to types and functions (through compile-time function evaluation), the same language is used for both the application expression *and* generating parameters. Thus, the Hierarchical Transactional Language (HTL) is both a language and a meta-language using precisely the same semantics and syntax.

4.4 Syntax

This section will introduce the HTL syntax, providing small demonstrative examples.

Typographical Conventions Typographical formatting uses `sans-serif` to describe variable identifiers, rule identifiers, token identifiers, type names, and keywords. As a matter of convention, tokens are identified with a lowercase "t" followed by a capitalized semantic name. Rule and class identifier are capitalized,

while variables are generally all lowercase (with the exception of large arrays and iterators, which use an uppercase letter convention). The designer is free to use any legal identifier that suits their specification style.

Whitespace Scope Whitespace determines scope blocks. A tab character, or a series of spaces coalesced, determine a lexical INDENT token in the lexer.

Scoping Rules Scope is syntactically defined, and constrained to source files. Each source file is treated as a module containing rules and classes. If a given rule in file `A.hr` instantiates a rule in `B.hr`, they remain separate scopes. This *ruletree/source* scope preserves readability. In any case where scopes need sharing, they are done so explicitly to communicate intended and expected behavior.

Identifiers All rules, variables, types, tokens, and classes are represented with unique identifiers. Identifiers must begin with a letter or underscore, followed by zero or more alphanumeric or under characters thereafter.

Rule Identifiers/Rule Scope Rule identifiers are scoped to their parent. Identifiers may be duplicated for different trees; however, the compiler maintains a global rule identifier list for a given source file. It issues toggleable warnings in case of a name collision; this enforces unambiguous identifiers.

Variables Variable identifiers are scoped to their parents, and every descendant of the parent in the same source file. Variables follow ruletree/source scope. Rule instantiation (via a `call` statement) delineates a new scope.

Variables are declared with the `var` statement, followed by a list of variable identifiers, a colon, and a type instantiation. If multiple identifiers are provided, each one receives its own type instantiation – it is equivalent to separate `var` statements for each identifier.

Types Type instantiations consist of the type name (i.e. one of the built-in types or a user-defined class), and its specific parameters in order of the declaration. All types must be statically resolved at compile time – parametric designs are realized by allowing variables in the type declaration under the proviso that the variable's value is constant and determinable by the compiler.

Tokens Tokens are declared implicitly by use. A `create` statement declares a token. Any token guards containing a token with the same identifier will have that token in its guard. Note that token scoping is different from variable scoping: they are semi-global to the rule tree in which they're created (again, limited to the source file).

Tokens can only be passed up the hierarchy. The compiler reports an error when a token is created by a rule and guarded by the child of one of its peers, as that behavior is undefined.

4.4.1 Rules

A rule consists of the `rule` keyword followed by an identifier, and an optional guard. It then contains a statement block with either variable declaration and child rules, or variable declarations and atomic statements. The two cannot

be mixed – rules can either be hierarchical containers or leaf nodes with state-modifying actions.

The simplest rule contains no guard and no statement (the `pass` statement indicates no behavior):

```
rule Foo:
   pass
```

A rule's guard can be a Boolean combination of token identifiers expressing the condition under which it can fire:

```
rule Bar(tX and tY):
   pass
```

In this case, rule `Bar` will wait for *both* tokens `tX` and `tY` before it executes.

If no guard is present, then the rule may begin whenever its parent begins (potentially after some initial latency). In the following example, child rules `B0` and `B1` may run in any arbitrary ordering; since there are no dependencies expressed, any ordering is assumed valid.

```
rule B:
   rule B0:
      x = 1
   rule B1:
      y = 1
```

Tokens can only be passed among parents and ancestors. The following demonstrates a valid hierarchical token transfer:

```
rule Top:
   rule Child1:
      rule A:
         create tA
   rule Child2(tA):
      create tB
```

In this example, rule `Top` begins, then `Child1` executes. Its child, `A` begins and creates token `tA`. Since none of `Child1`'s children contain `tA` as a guard, and since there are no more outstanding rules to execute, `Child1` terminates – however, it

commits token `tA` to its parent `Top`. The token is then passed to the guard of `Child2` which may now execute.

The following is an example of an *invalid* token transfer:

```
rule Top:
   rule Child1:
      create tA
   rule Child2:
      rule B(tA): # illegal
         create tB
```

This is undefined behavior and will cause an error. After `Child1` executes, it would need to pass `tA` to `Child2`. However, without a dependency between the two, `Child2` may execute *before* `Child1`. Further, even if `Child2` executes after `Child1`, the implicit dependency makes the code unintuitive and difficult to follow.

4.4.2 Atomicity

Rule bodies are atomic. They only need to be ordered due to naming declarations – e.g. create statements must come before the token's use in a guard, encouraging human-readable code.

Any conflicting writes follow the same rules as Verilog. The last (in source code order) of the valid assignments are accepted. Consider the following rule:

```
rule T:
   x = 1
   x = 2
   x = 3
```

Only the last of the three assignments ("`x = 3`") is synthesized. The other two assignments are removed.

It is important to bear in mind that atomic statements mean that ordering is important only for resolving ambiguous concurrent assigns.

```
rule T:
   if x > 1:
      y = 1
   y = y + 1
```

All of the references to `y` on the right side of assignments are the *same value* – they are the value that was atomically read in at the beginning of rule `T`. The values on the left sides refer to the value being written. The above example is equivalent to:

```
rule T:
   if x > 1:
      y_out = 1
   y_out = y_in + 1
```

Thus, the way to interpret this is that the conditional block is obviated, as it will always be overridden by the second assignment.

4.4.3 Conditional Statements

Conditional statements are very similar to Python: an `if` keyword is followed by a condition, a colon, and a block statement. It may be followed by `elif` statements and a final optional `else` statement.

Internally, the compiler rewrites statements into a declarative form, coalescing conditions into independent atomic statements.

For example:

```
if x:
   if y and z:
      create tA
   elif w == 1:
      create tB
   else v == 2:
      create tC
   create tD
else:
   create tE
```

is internally translated into the following independent atomic statements:

```
x and (y and z) -> create tA
x and not(y and z) and (w == 1) -> create tB
x and not(y and z) and not(w == 1) and (v==2) -> create tC
x -> create tD
not x -> create tE
```

The latter format is ideal as an intermediate, compiler-interpreted language as it simplifies and decouples `create` statements. For many designers, however, it is unintuitive and difficult to extract its high-level meaning, hence a natural and familiar `if/else if/else` syntax.

4.4.4 Rule Instances

Design reuse is pivotal to ensuring ease of IP integration, specification longevity, and amortizing testing effort over multiple applications. Functions in sequential software and modules in HDLs serve the same fundamental role: they allow behavior to be self-contained (thus making the specification managable), and allow reusability. Further, they provide designers a way to intuitively break down an application into its subparts, making the code easier to read, maintain, and isolate problems in. While the transaction hierarchy provides this functionality, it would be cumbersome and unreadable to specify full designs within a single, deep hierarchy. Rule instances allow easy code management and organization, parameterized rules, and provide an easy-to-understand reusability mechanism.

4.4.4.1 Rule Instance Syntax

Rule instances use the `call` statement followed by the name of the rule and input arguments. Since the token naming scope is semi-global, token namespaces between caller and callee are kept separate. Otherwise, the token namespace

would become polluted causing token identifiers to collide. For example, if the rule `process_data` is called and it creates a token internally named `tDone`, and the caller also contains a token with the same name, it would likely raise compiler errors due to poor token scoping.

For example, consider the following:

```
rule Top:
   rule setup:
      create tDone
   rule caller(tDone):
      call Foo

rule Callee:
   rule do_some_work:
      create tDone
```

To resolve this, all calls are optionally block statements containing a token identifier map via the `creates` keyword (explained further below). Control tokens can now pass from callee to caller, but only through explicitly defined means.

Arguments Input arguments are passed either positionally or by name, following Python's syntax rules. While optional arguments are not yet allowed, named arguments provide the benefit of clear code documentation

```
call Callee(arg1, arg2, named_arg=arg3)
```

Return Values Callable rules may return a value via a `return` statement followed by the expression being returned. Type is inferred, and all callable rules are untyped by default. In the following example, rule `add` is called with arguments `a` and `b`. The return value is the summation of the two input arguments.

```
rule add<-(x, y):
   return x + y
call sum = add(a, b)
```

Note that callable rules are full-featured and follow precisely the same semantics as non-callable rules. In reality, the compiler makes no internal distinction – it simply tags certain rules with argument requirements. To facilitate flexible and parametric design, arguments may be variables, expressions, or type templates. In the following, rule `foo` receives a type template for a `Bits` type (with bitwidth of 32). Within `foo`, variable `intermediate` instantiates its type based on the provided template. The compiler will infer types based on use, and thus there is no need to provide any extra information about the argument.

```
1  rule foo<-(type, x, y):
2     var intermediate: type
3     rule setup:
4        intermediate = x + y
5        create tNext
6     rule next(tNext):
7        intermediate += 1
8        return intermediate
9
10 call v = foo(Bits(32), a, b)
```

Token Maps Token maps are optional statements contained within call blocks that specify the tokens to export from the callee. Additionally, the designer may rename tokens to prevent namespace pollution.

```
1  call Callee(..):
2     creates tDone->tCalleeDone, tError
```

4.4.5 Functional Rule Instances

A functional rule instance returns a value, and can be treated as a preamble. If rule `A` calls functional rule `foo`, then `foo` is an instanced rule that executes before rule `A`'s body. Thus, it reads the same data that `A` reads. The return value of the

functional rule is made available to `A`. Note that all of this behavior is implicit – the compiler manages ordering and correct rule execution.

Consider the following code:

```
rule Top:
   if add(1, 2) > 1:
      create tCorrect

rule add<-(x, y):
   return x + y
```

Internally, the rule executes identically to the following code:

```
rule Top:
   var r: Bits(2)
   rule do_add:
      r = x + y
      create t_do_body
   rule do_body(t_do_body):
      if r > 1:
         create tCorrect
```

4.4.6 Classes

Managing design complexity is eased through abstraction models. Arguably the most common such model is the class object – it packages state and behavior such that internal intermediate actions do not need to be exposed to other components in the design. Classes in HTL specify a user-defined type and may contain internal parameterizable data along with rules to modify that state. They encourage object-oriented and help organize large applications into logical subcomponents.

Compared to other languages, HTL classes must follow certain rules in order to remain compatible with transaction semantics. When a class object is read by a rule, it is read-in as a whole. Likewise, when it is committed, it is committed as a whole.

Classes may contain parameters that conform to "duck typing." As long as the compiler is able to determine a parameter's type statically, the code is valid. There are no private variables – all variables in a class are public. It is up to the developer to enforce reasonable access rules, following Python's access semantics.

4.4.6.1 Instantiation

A class is instantiated by declaring a variable with its type. Consider the following code:

```
1 class Foo:
2    var x, y: Bits(16)
```

Class `Foo` is instantiated through a declaration, such as:

```
1 rule Top:
2   var f0: Foo()
```

4.4.6.2 Parameters

Parameters are passed to the class at type instantiation. Below, `Foo` is extended to include a `bitwidth` parameter:

```
1 class Bar<-(bitwidth):
2    var x, y: Bits(bitwidth)
```

And instantiated using:

```
1 rule Top:
2    var b0: Bar(32)
```

Note that the parameters are untyped – instead, their type is determined valid if they are used correctly. In the example, an integer is a valid type for this specific instance of the class, since it is a valid type for the `Bits` class.

Not only can type parameters be arbitrary types, they may also be used as type *templates*. A type template simply passes an uninstantiated type along with

its parameters. Below, the same class is now made generic to accept any type template:

```
class Baz<-(T):
   var x, y: T
```

A valid instantiation must provide a full type instance.

```
rule Top:
   var bz: Baz(Bits(24))
```

When the compiler encounters a type declaration, it creates an internal type template. This may be passed as a parameter, or it may be turned into an instantiation if it appears in a variable declaration statement. Clearly, an integer type parameter (`var bz: Bar(16)`) would cause an error: `16` is not a valid type template.

Type parameters must be statically resolvable when the compiler reaches the variable declaration. Consider the four declarations below:

```
var x: Bits(16)
var y: Int(20)
var z: Bits(y) # Ok: compiler knows y is a constant value
var v: Bits(x) # Not statically resolved
```

An `Int` object is immediately tagged as static (they are read-only), and thus the declaration for `z` is valid. The declaration of `v`, on the other hand, is illegal: its bitwidth cannot be determined at compile time.

4.4.7 Parametric Class Examples

Constant analysis enables parametric types – the following examples demonstrate a container class for complex numbers and a simple matrix class containing a scalar rule.

4.4.7.1 Complex Class

```
class Complex<-(type):
   var real, imag: type
var C: Complex(Bits(16))
```

During object instantiation of `Complex`, the type instance `Bits(16)` is passed as a parameter with the identifier `type`. In the body of the `Complex` class, two variable declarations `real` and `imag` are issued the `Bits(16)` type. In this particular case, the type parameter can be any available type. If, however, the owner of the `Complex` instance attempts to perform addition and multiplication on the class's members, and they do not support the operation, the type checker will raise a compiler error.

4.4.7.2 Matrix Class

```
class Matrix<-(type, N, M):
   var mem: Array(type, N*M)
   rule scale<-(self, scalar):
      for x in IRange(mem):
         x *= scalar
# ...
var M: Matrix(Bits(32), 16, 16)
# ... operations to load the matrix ...
call M.scale(1.0)
```

Similarly to the `Complex` class, the `Matrix` object creates an member of type `Array` whose elements are typed `Bits(32)`. The size of the array is computed statically and a call to `scale` executes a simple iteration where each element is multiplied by the scalar.

Clearly, the code is concise and legible at first glance without sacrificing performance simply by restricting specification expression power.

4.4.8 Modifiers

Pragmatic statements communicate certain semantic information to the compiler. Modifiers use the following syntax:

```
@modifier_name(arg0, arg1, ...)
rule foo:
  pass
```

They are particularly useful for iterators, discussed in the next section. Often, in order to preserve static data types, it is important to force an iterator to unroll and resolve its data type. Modifiers used in the Fast Fourier Transform example (Appendix B.1) to specify how information passes between stages and to ensure that the types are statically and correctly resolved.

4.5 Compiler

The Hierarchical Transaction Scheduler and Synthesizer (HTSS) compiles and simulates HTL. It includes a rich type and iterator library, race condition detection mechanism, a static analysis engine to allow design parameterization via inferred constant variables, scaffolding synthesis (internal bookkeeping such as token dependencies and variable read/write cataloging), a high-level transaction-level behavioral simulator, and a synthesis back-end.

Integration of all of these components, as well as associated compiler-specific options (e.g. whether to unroll certain iterators) is aided through a built-in TCL parser. Design scripts can be written to simplify and automate the overall design flow. While the initial implementation is simple, a built-in scripting engine has the

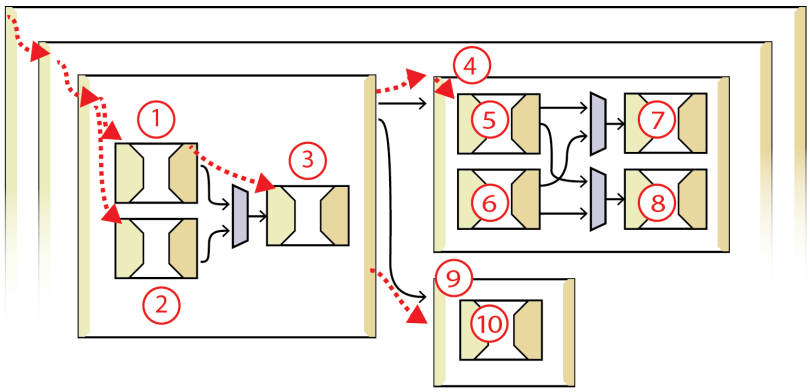


Figure 4.2: Rule tree traversal order – it dives into the child rules first, visiting them in topological order based on their dependency. This guarantees that by the time a rule is visited, all of its predecessors have already been visited.

capability of passing constant values as inputs to top-level rules, thereby allowing parametric sweeps and automatic architectural discovery.

4.5.1 Parsing

The compiler uses an ANTLR-based parser generator to generate abstract syntax trees (ASTs). Next, it constructs transaction-scale hierarchies, each containing a local AST. It further catalogs all variable declarations and their types (although it does not do type instantiation at this phase).

Once the foundations are set up, the compiler traverses the tree, and from the bottom-up topologically visits each rule. That is, a rule-tree traversal climbs down the hierarchy. At the leaf rules, local AST walkers analyze variable declarations (for type instantiation), conditions, assigns, “create” statements, and call statements.

Next, a local dependency walker iterates through the following visitors:

- Variable Elaborator: Catalogs all reads and writes for the rules (including all descendents)
- Token Analyzer: Constructs dependencies, appropriately tagging conditional tokens (for later use in closure analysis).
- Type Analyzer: Instantiates types for all of variables for this rule. Note that it can leverage any incoming static reads/writes, since it assumes full analysis has already completed for its predecessor rules.
- Constant Analysis: Finally, catalog all write statements to determine which, if any, variables are written with constant values.

Variable Read/Write Elaboration The simplest, first-cut optimization involves trimming read and write copying where they are not used. If transaction `Parent` contains variables `x, y, z`, and a child transaction `A` only reads `y`, then there is no reason to copy `x` or `z`.

Token Analyzer Token scoping and guard enforcement ensure valid, constructable guards for simulation and synthesis.

Type Analyzer The heart of the inferred type engine collects all known constant variable information and creates type instances for each variable, summarizing their specific, resolved type parameters.

Constant Analysis Once all of the writes have completed, this module forward propagates constant-value information.

4.6 Static Analysis

Unlike traditional sequential software, self-contained data/control scope guarantees behavior that allows straightforward static analysis. Access to state is always controlled and well-defined. The compiler can quickly determine if a variable remains constant throughout its data path. If that variable is later used to conditionally generate a control token, unreachable token paths (and their rules) are eliminated. Since there is no difference between a dynamic variable and one that is detected to be constant, designers can leverage this language feature to yield powerful design parameters, much in the same way they do with HDLs.

In the following example, rules `A` and `B` are effectively meta-rules that produce a type parameter for rule `C`, via the `w` variable.

```
1  rule T:
2     var x, y: Bits(32)
3     rule A:
4        x = 1
5        create tA
6     rule B(tA):
7        y = x
8        create tB
9     rule C(tB):
10       var w: Bits(y)
```

4.6.1 Static Analysis for Constant Inference

At compile time, variable access is analyzed to determine if the value remains locally constant. That is, for a given rule, the compiler tracks inherited and

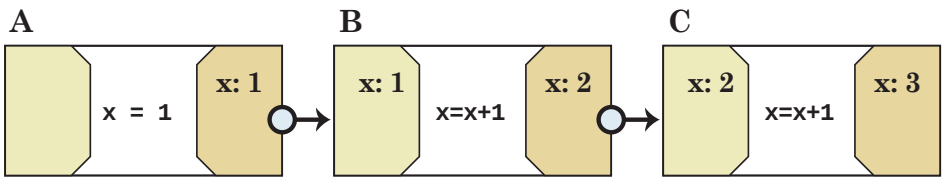


Figure 4.3: Serial rules with local constant values

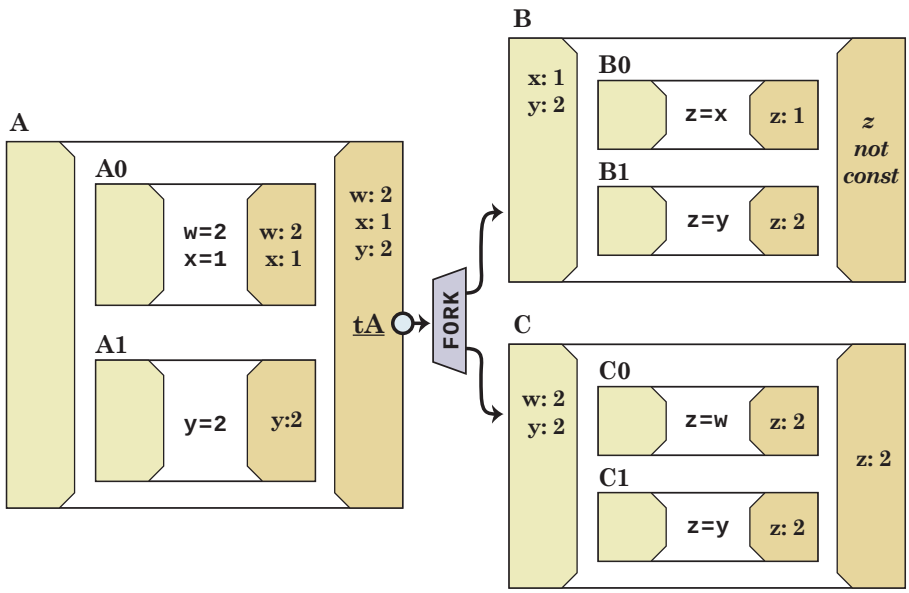


Figure 4.4: Constant inference demonstrating rule-level parallel writes.

owned variables to determine if they hold a constant value for the local rule's scope (independent of any inputs to the system).

Figure 4.3 illustrates local constant values. Consider three serial rules – the static analyzer is able to determine local constant values for `x` on a per-rule basis. This is particularly useful if, for instance, a forward rule dependent on `B` uses `x` as a type parameter.

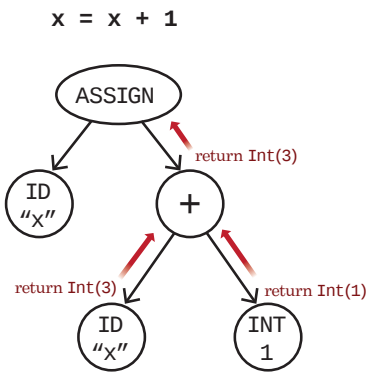


Figure 4.5: AST-level constant value inference – assume that the rule is provided `x` with a constant value of `Int(3)`. The AST walker visits each node hierarchically and returns an evaluated result for the sub-tree. If the subtree is not constant (dynamic), then it returns a NULL value.

Recall that the static analyzer traverses in topological order. There are two parts to constant inference. The first happens at the AST level. The algorithm visits assignment statements and conditionals.

4.6.1.1 Local vs. Global

Local constancy simply means a variable holds a constant value for its rule and its child rules only. Global constancy applies to every use of that variable.

4.6.1.2 AST-level Constant Inference

Assignments An AST walker traverses a rule's body and attempts to determine the dynamic or constant nature on a per-node basis. Fig. 4.5 illustrates a simple addition statement. In this case, the statement will take the value of `x` read in from its parent, increment it, and commit the incremented value. At the top-

level, the walker visits the ASSIGN node. It recursively dives into the right value (RVALUE) node to visit the addition node. This recursively calls its children, starting first with the IDENTIFIER node for `x`. Assume that the parent tells this child that `x` holds a constant value of 3. The walker will instantiate an `Int` object with the value 3 and return that. Next, the constant literal value of 1 is visited – again, it will instantiate an `Int` with value 1. Back at the addition node, the walker assumes that the typechecker has already ensured that this is a type-valid operation. It takes the `Int(3)` object and executes an addition with the `Int(1)` object to yield a new `Int(4)` object. Back at the ASSIGN node, the walker now sees that the RVALUE has returned a constant value and stores this value into the rule's write table.

Conflicts If there are multiple assigns to the same value, then only the last write is considered valid. The following will still treat `x` as a constant value of 2. The first assignment statement is obviated since these are deemed to execute atomically. The simulator issues a warning to notify the developer when this occurs.

```
1 x = 1
2 x = 2
```

Code Fragment:	If w is **not** constant:	If w is constant:

`if w > 1:`
`  a = x`
`else:`
`  a = y`
	a is constant only if:
	If x and y are both constant, and have the same value.
		If w's static value > 1:
		a is constant only if x is constant
		If w's static value ≤ 1:
		a is constant only if y is constant

Figure 4.6: Constant inference for "if" statement example

Conditionals At every "if" statement, the condition expression is evaluated to determine if it has a constant value. If it does, then the appropriate branch is isolated and visited, and the AST subtree associated with the branch is tagged accordingly (that is, if the expression evaluates to True, then the False branch is marked as "compile-time obviated"). If the condition cannot be evaluated, then it visits all branches (including "else if" and "else" branches). Each recursive call into the respective branches returns a mapping from variable to constant value. The compiler cross-correlates all of the maps and if for every map, a given variable holds a constant value, then that variable is determined to hold a constant value for the entire rule.

Consider fig. 4.6. If `w` is not constant, then both branches must be analyzed. The only situation in which `a` is constant is if both `x` and `y` are constant and have the same value. If any of those conditions are violated, then `a`'s value cannot be inferred and it will remain a dynamic variable. In the second scenario, if `w` *is* constant, then the branch value can be determined at compile time. If $\texttt{w} > 1$, then `a` will hold constant only if `x` does. Likewise if $\texttt{w} \leq 1$, `a` will hold constant only if `y` does.

4.6.1.3 Rule-level Constant Inference

Once the leaf rules' internal AST nodes have been analyzed to collect the rule's constant value writes, a rule-tree traversal (again, in topological "forward" order) combines the constant values for multiple rules.

4.7 Simulator

The compiler includes a fast simulator amenable to print-based debugging, along with explicitly verbose execution. It maintains a queue of transactions, and dynamically allocates both the space and execution interpreter for each transaction. Tokens are stored within their parent transactions and committed using an event-driven dynamic scheduler.

Induced Delays Since the simulator essentially selects a single path of execution through all possible executions, timing variation coverage is addressed through manual injection of specific timing delays. Monte Carlo simulation through random latency injection provides a method of stochastically exploring latency variations. Paired with the conservative exact race condition detection in sec. 6.2.5, the two tools form a powerful method of extracting concurrency bugs.

Chapter 5

Iterators

The philosophy behind HTL is to constrain input specification rather than limit the synthesis tool. An egregious source of synthesis constraints in high-level synthesis tools stems from arbitrary access to large, indexed state (i.e. memory). Compared to traditional specification formats, HTL guarantees high performance array access only if designers describe indexed behavior through built-in iterators (see Chapter 8). This puts the burden on the designer to clearly define their intent, while opening opportunities for automatic realizations of dynamic hardware or software to manage otherwise complex indexing. To demonstrate that such a model is practicable, a Fast Fourier Transform (FFT) is implemented through composition of built-in iterators.

5.1 Motivation

Practical, real-world applications require a method for representing large collections of related information, e.g. an image, a matrix, etc. Vectored data and matrices form the core of practically all signal processing applications, and provide

a familiar mechanism for representing common algorithms. The copying semantic, however, complicates how large state overhead is mitigated during synthesis. If the indexes accessed are known, then only those values are copied. If they are not known, however, the compiler would have little choice but to institute copying (or equivalent synchronization/caching architectures) for the entire array. For any medium to large array, this is impractical.

In general, arbitrary array access hinders synthesis optimization opportunities. It forces conservative worst-case behavior to ensure correct execution, since the synthesis tool does not have enough information to ascertain which portions of the array may be needed. This leads to unnecessary state copying and transfer.

5.2 The Array Primitive

HTL takes a unique approach: it provides an `Array` library primitive with the proviso that arbitrary indexing is explicitly disallowed on this object. Instead, elements of an array are selected by composing built-in, well-defined finite-state iterators. Besides solving the copying problem, iterators alleviate data access uncertainty, avoiding potentially cumbersome and difficult verification. They guarantee lifetime bounds on state, allow easy partitioning of array access, and avoid common pitfalls in verification.

Iterators translate well to other approaches to cosynthesis when converted into data-flow graphs. They guarantee well-formed graphs, allowing designers to leverage any of the existing mature high level synthesis packages. The tools do

not need to apply loop unrolling or analysis heuristics, as the graphs are directed and acyclic. They can thus be directly scheduled.

When indexing is absolutely necessary (and the designer is willing to sacrifice potentially significant performance), indexing is allowed via an `Addressable` library primitive. This object is provided under the assumption that the designer use it responsibly and with knowledge of the resultant performance consequences.

5.3 Communicating Designer Intent

In addition to their incompatibility with the copying behavior, variable array indexes are antithetical to the philosophy of clear expression of designer intent. Arbitrary arithmetic index expressions requires symbolic analysis that is very difficult and scales poorly.

For instance, the question of selecting subsets of the array is nontrivial. If access to the array occurs in spans, then the compiler must analyze how data is accessed and attempt to partition or replicate subsets of the array accordingly. Through an iterator, however, this information is directly conveyed to the tool – there is no need for arithmetic or symbolic analysis of index variables.

5.4 Execution Semantics

In software contexts, an iteration implies a specific ordering on a data set. In HTL, the definition is split into two parts. Iterators specify complete or partial orders on the *dependencies* between array elements (either within a single array

or, more commonly, across two different arrays). The *execution*, however, is not necessarily beholden to the order in which the iterator generated its dependencies; it simply respects the dependencies induced. Sec. 5.5.4 clarifies this difference with an example.

5.5 Syntax & Semantics

Syntactically, an iterator uses a `for` statement providing an iterating variable identifier along with the iterator object. Semantically, these still define transactional behavior. Each iteration is a new transaction; it contains copies of its parent's state, with the exception that it does not copy the source array. It only receives its iteration value(s).

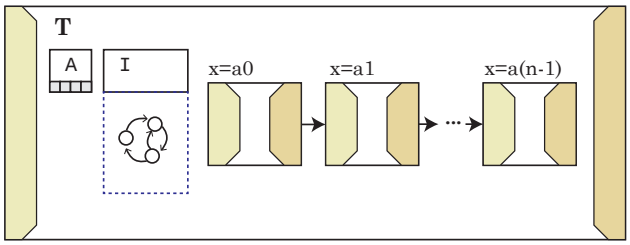
5.5.1 Token Scoping

Tokens may pass between rules within an iterator. However, every iterator is treated as an independent token scope. External tokens are created in the `finally` block.

```
1 for x in Y:
2    # ...
3 finally:
4    create tDone
```

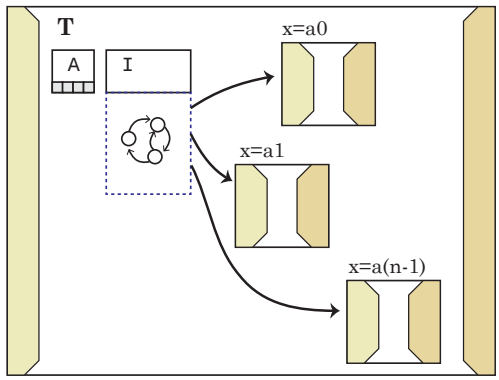
Semantically, the `finally` block is a rule that executes after all of the iterations have completed – it is the closure of the full iteration.

```
rule T:
  var A: Array(Bits(16), 128)
  var I: ISequence(A)
  for x in A:
    pass
```



(a) Sequential iterator execution.

```
rule T:
  var A: Array(Bits(16), 128)
  var I: IRange(A)
  for x in A:
    pass
```



(b) Concurrent iterator execution.

Figure 5.1: Illustration of execution semantics between sequential and concurrent iterators.

5.5.2 Sequential vs. Concurrent Iterators

Iterator execution is either sequential (ordered) or concurrent (unordered). If it is ordered, tokens are generated between iteration stages (which, again, are simply transactions). Otherwise, the iteration bodies themselves may execute in any order. Fig. 5.1a illustrates a sequential iterator – all but the first iteration bodies are dependent on the previous iteration. Contrariwise, fig. 5.1b illustrates concurrent iterators – there are no direct dependencies between iteration bodies.

5.5.3 Multiple Iterators through Co-iteration

Consider an application which needs to iterate an input array and generate an output in a different sequence. HTL allows co-iteration of multiple iterations via a special double pipe (`||`) composition operator. At every iteration, each of the iterators will provide their iterating value. These values are collected and passed onto the child transaction (the iterator body).

5.5.4 Ordering of Iterator versus Iterator Execution

Consider the following example.

```
1 var X: Array(4, Bits(8))
2 for x in IAlternate(X, 0) || y in IAlternate(X, 1):
3     x = y
```

Assume an iterator `IAlternate` selects every alternative element in the array, offset by the second argument provided (`0` for the iterator associated with `x` and `1` for the iterator associated with `y`). If the array `X` contained `[0, 1, 2, 3]`, then the above code would be equivalent to the following rules:

```
1 rule FirstIteration:
2   X[0] = X[1]
3 rule SecondIteration:
4   X[2] = X[3]
```

Assume that `IAlternate` executes unordered – that is, it does not induce control dependencies between iteration body transactions. The first iteration may execute after the second. In its *execution*, the iterator is partially ordered. However, the actual index *dependencies* generated by the iterators are completely ordered. It is not the case that, for instance, `X[2]` will get assigned to `X[1]`.

Another way to interpret this distinction is between compile-time iteration and run-time iteration. This view is useful for understanding the concept, but is

problematic insofar as it assumes that iterators are fully unrolled at compile-time. This is not the case – iterators communicate state machines that can dynamically produce indexes and thus can be synthesized accordingly.

5.6 Built-in Iterators

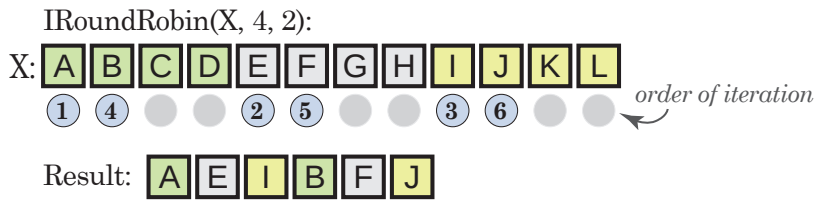
Fig. 5.2 illustrates three iterators used for the FFT example below. They are concurrent (unordered) iterators. This does not mean the array is unordered; in fact, these iterators yield each iteration value based on the original order of the array itself. However, there are no control dependencies between iterations, and thus they execute concurrently.

5.7 FFT through Iterators

The Fast Fourier Transform (FFT) is a widely used algorithm that exhibits complex indexing and is a prime example of the power of iterators. This is demonstrated through a decimation-in-frequency FFT implementation composed of iterators, illustrating the robustness of the language's iterator semantics. Recall that each `for` statement follows transactional, hierarchical semantics. Every iteration is a new transaction following the copying mechanism, but rather than copying the entire array (in this case, of inputs and coefficients), the iterators isolate specific elements.

Figure 5.3 lists the heart of the FFT application. Input parameters `size` and `type` are provided. The preamble sets up initial stage count calculations, and

IRoundRobin(array, length, limit)
Skip every 'length' element, until 'limit' cycles have been iterated.



IClustered(array, cluster_length)
Provide 'cluster_length' elements in groups each iteration

IGroupedRoundRobin(array, rr_length, rr_limit)
Like IRoundRobin, but provides elements from each group at the same time.

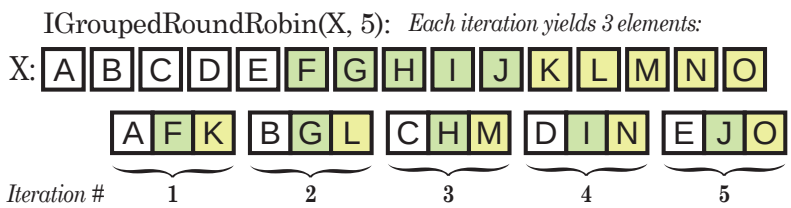


Figure 5.2: Built-in iterators provided to allow FFT indexing

```
1  var stages: Int(math.log2(size))
2  var i_stages: ISequence(stages)
3  var intermediates: Array(size, type) # intermediate results
4  @pipeline(X, intermediates)
5  for stage in i_stages:
     # Calculate the skip size for this stage
6    var cluster_size: Int(math.pow(2, stages-stage))
7    var rr_size: Int(math.pow(2, stage))
     # Instantiate iterators
8    var in_clustered:   IClustered(X, cluster_size)
9    var coef_clustered: IClustered(Coefficients, cluster_size)
10   var out_clustered:  IClustered(intermediates, cluster_size)
     # Iterate over the sub cluster
11   for in_cluster in in_clustered ||
12       c_cluster in coef_clustered ||
13       out_cluster in out_clustered:
       # Now iterate within the group
14     var i_in:   IGroupedRoundRobin(in_cluster,  rr_size, rr_stride=2)
15     var i_coef: IGroupedRoundRobin(c_cluster,   rr_size, rr_stride=1)
16     var i_out:  IGroupedRoundRobin(out_cluster, rr_size, rr_stride=2)
17     for out0, out1 in i_out || x, y in i_in || coef in i_coef:
18        out0 = x + coef*y
19        out1 = x - coef*y
```

Figure 5.3: Code listing of FFT with Iterators. Each iteration is a new transaction. The double-pipe operator composes parallel iterations – elements are selected from the iterators specified and passed to the iteration transaction at the same time.

defines the outermost stage iterator. Note the "`@pipeline`" modifier causes the static elaborator to unroll the iterator and compile time, copying information from one stage of iteration to the next. This iterator requires unrolling since intermediate data (`in_clustered`, `coef_clustered`, and `out_clustered`) is dependent on the current iteration value (i.e. the cluster size).

Fig. 5.4 illustrates an 8-point decimation-in-frequency FFT, labeled with the source indices for each stage. The iterators from lines 11-13 grab a sub-array of elements via the `IClustered` iterators. The iterator values yielded (which are themselves `Array` objects) are then iterated upon via round robin iterators (lines 14-16).

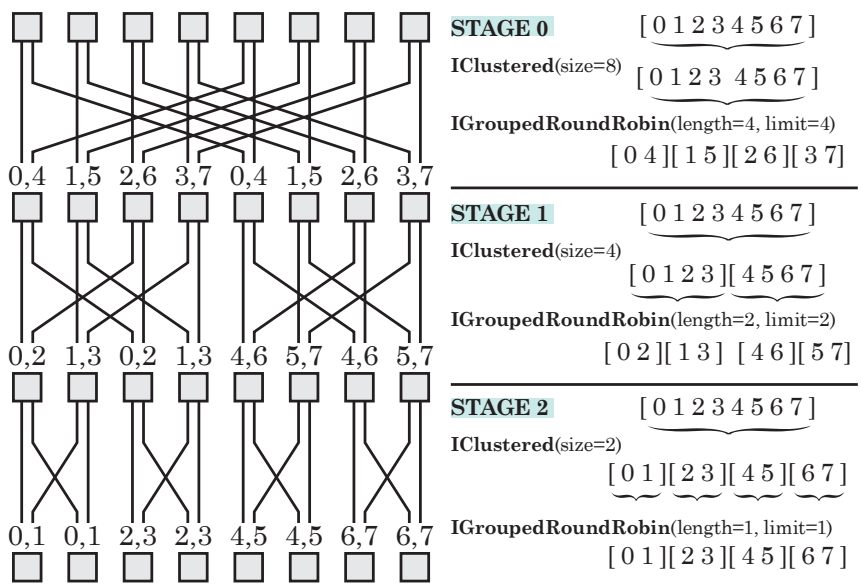


Figure 5.4: FFT indexing achieved through iterators

Chapter 6

Practical Analysis and Model Translation

This chapter details practical design analysis of hierarchical transactions. The first concern is a closure mechanism. In the semantic model, a transaction completes once all of its children that can run have completed (i.e. any child that has had its guard satisfied). An open-ended implicit condition allows domain-appropriate implementations. Realizing closure in software versus hardware lead to rather different solutions.

The second concern is race condition detection. The combination of latency tolerance and concurrency leads to potential operational ordering sensitivity which can lead to deadlock. A scalable, conservative method for identifying possible race conditions is presented and discussed.

Finally, in order to demonstrate that hierarchical transactions are in fact capable of efficiently translating to existing models, this chapter outlines methods of translation to pure functions and control/data-flow graphs that can leverage existing high-level synthesis solutions.

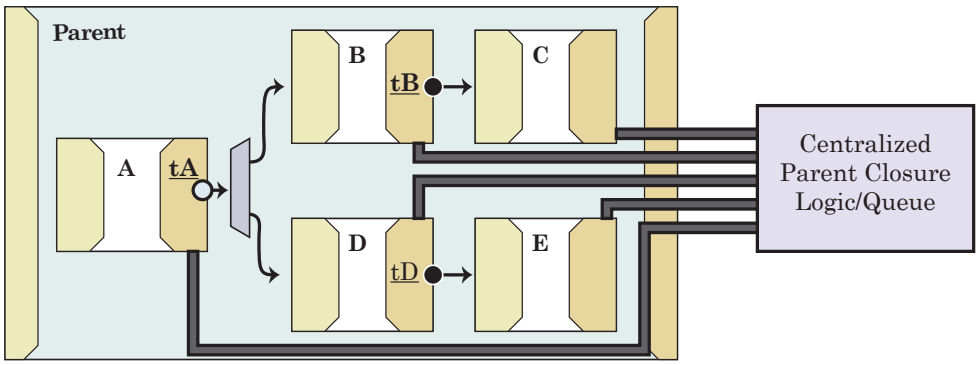


Figure 6.1: Centralized control to determine parent transaction termination

6.1 Closure & Distributed Transaction Control

Transactions can be directly executed for simulation, verification, and (when managed correctly) implementation. Improving performance of all three enables rapid development – faster simulation translates to shorter development time.

During direct execution, the issue of when to terminate a parent transaction needs to be addressed, referred to as its *closure condition*. The simplest approach uses a central controller with full knowledge of every child transactions state, along with a queue of outstanding transactions. Fig. 6.1 illustrates the inserted logic required to determine when transaction PARENT completes – namely, it must observe the state of all transactions. If all transactions have completed, and there are no outstanding transactions in the queue, then the parent transaction may terminate.

While this approach is correct, it may not be suitable when executing transactions in e.g. hardware where a queue is not feasible due to resource constraints. Instead, a more efficient approach determines all possible conditional execution

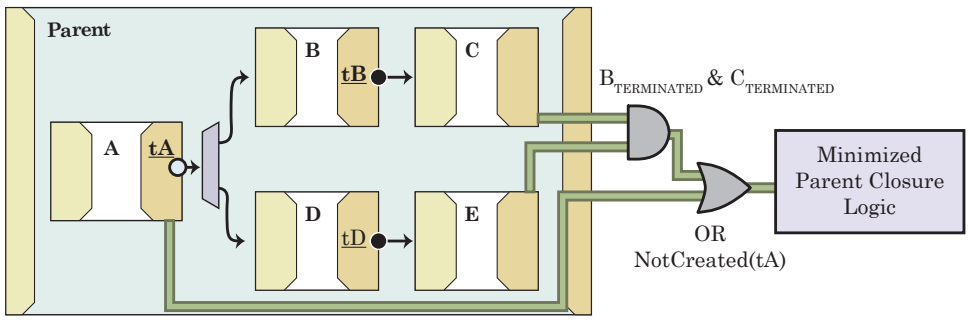


Figure 6.2: Centralized control with synthesized logic to determine parent closure.

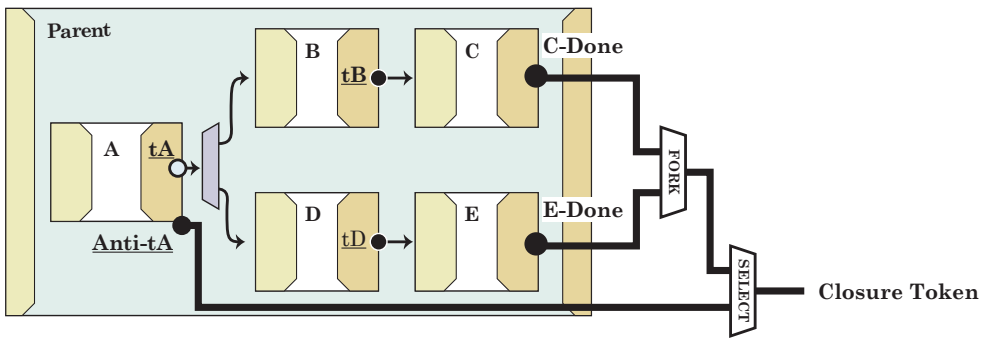
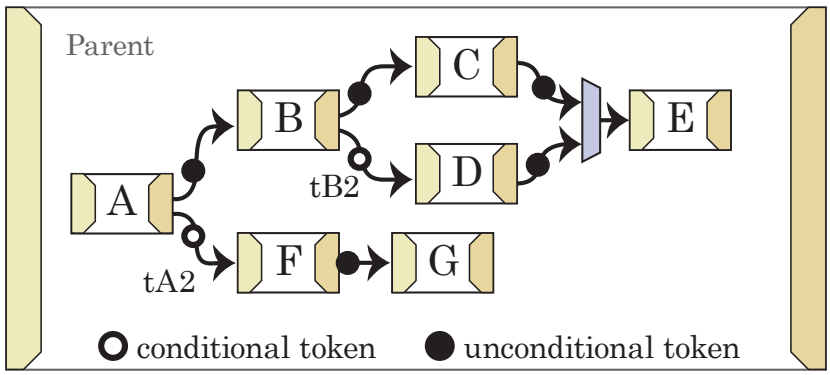


Figure 6.3: Distributed control using counter tokens to implement synthesized closure logic.

paths and synthesizes logic accordingly. In the example, transaction `A` creates a conditional token `tA`. Synthesized logic would check for one of the following conditions: either token `tA` was never created, or token `tA` was created and both transactions `C` and `E` have terminated. Fig. 6.2 illustrates the logic generated. This direct translation couples transactions in the target implementation (with, for example, logic wires), and thus may induce undesirable synthesis constraints. In the case that the transactions exist within different domains, this can affect performance and timing closure.



Closure Condition:
(E or NotCreated(tB2)) and (G or NotCreated(tA2))

Figure 6.4: Constructing the condition under which Parent can be closed, demonstrating how conditional tokens may terminate paths and must be accounted for.

The third approach generates *counter-tokens* or *anti-tokens* to pass transaction control state. Counter tokens are created at every conditional token creation – if the designed token is not created, then the counter token is created. A latency-tolerant, timing/locale decoupling signal now implements the same logic – that is, it replaces And with Join and Or with Select nodes.

Algorithm 6.1 lists pseudo-code for constructing the closure condition for a parent transaction with child transactions. It traverses the local child transaction network, starting with the "last" children (those without forward dependencies), first traversing backwards. It then visits each child in forward topological order: nodes' predecessors are processed before the node itself is processed. As it visits each transaction, it takes note of conditional tokens. If it is possible for the child transaction to *not* create a token, then that is a potentially *closing* transaction since it could terminate the path of execution. At a Join node, the algorithm

Algorithm 6.1 Closure condition construction algorithm for a parent transaction

```
1  def constructClosure(transaction parent):
2    condition = True
3    independent-children = {all children without forward transaction
         dependencies}
4    foreach child in independent-children:
5      condition = condition AND construct(parent, child)
6    return condition
7
8  def construct(transaction parent, child-node):
9    if child-node has already been visited:
10      return child-node.condition
11   if child-node is FORK:
12      condition = False
13      foreach predecessor P of child-node:
14         condition = condition OR construct(parent, P)
15   else if child-node is SELECT:
16      condition = True
17      foreach predecessor P of child-node:
18         condition = condition AND construct(parent, P)
19   else:
20      condition = False
21      foreach incoming token in child-node's guard:
22         condition = condition OR construct(parent, token-source)
23         if token.is-conditional:
24            condition = condition OR NotCreated(token)
25      foreach token created by child-node:
26         if token is conditional:
27            token.is-conditional = True
28   child-node.condition = condition
29   return condition
```

determines the condition under which execution cannot continue; this will occur if *any* of the incoming paths terminate. Its closure condition is the disjunction of all of its incoming paths' closure conditions. For SELECT nodes, on the other hand, execution can only stop at the node if *none* of its tokens are created. The conjunction of incoming conditions forms the node's closure condition. Since this algorithm visits each child transaction once, run-time is linear with the total number of transactions in the design.

Fig. 6.4 illustrates an example of closure condition generation. Below is a step-by-step runthrough of the algorithm. It starts from the transactions with no forward dependencies, transactions `E` and `G`. Starting down the `E` path, it visits predecessor `C`, which runs unconditionally.

Tracing the Closure Construction Algorithm Fig. 6.5 traces closure construction for the transaction hierarchy listed in fig. 6.1. Each colored box represents a recursive dive. As it visits each transaction, it analyzes conditional tokens and incrementally constructs the closure condition. The return values represent the net closure condition for a given node's predecessors. So, for instance, at node `B`, the return value from its recursive call is the total closure condition for all of `B`'s predecessors (in this case, just `A`).

6.2 Race Condition Detection

A new specification language brings with it the burden to provide verification scaffolding to ensure correct, intended application behavior. Early in the seman-

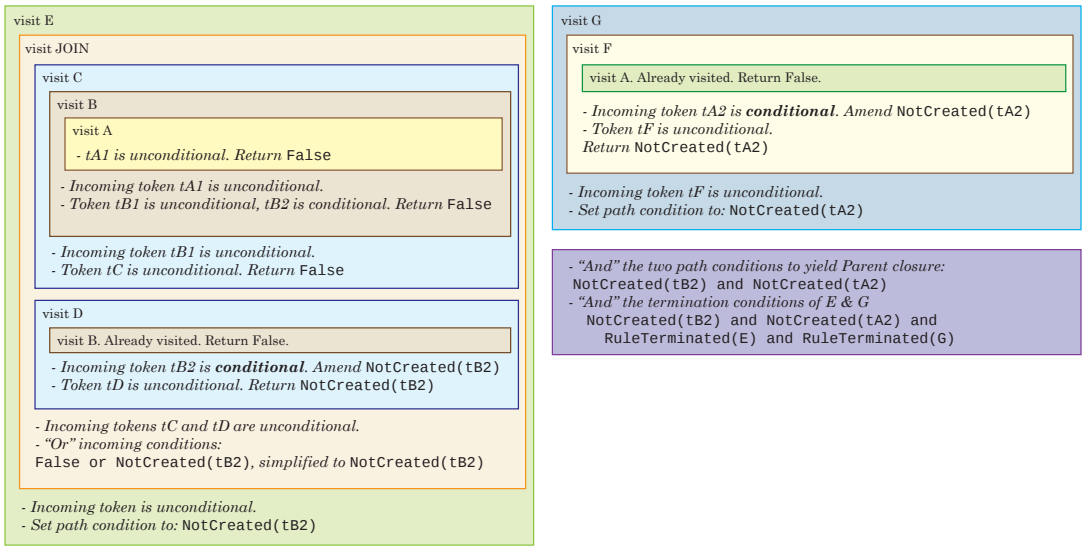


Figure 6.5: A trace of the closure algorithm

tic/language design process, it became clear that latency tolerance coupled with concurrency can quickly lead to correctness challenges. If any two transactions share information, and that information is written in an unexpected order, it may lead to starvation or deadlock. Consider transaction `A` and `B`, which both write a variable `x`. They execute concurrently and may write `x` in either the orders (`A.x`, `B.x`) or (`B.x`, `A.x`). A third transaction `C` is then executed, which conditionally creates a token `tC` based on the value of `x`. While this scenario is obvious to the designer, as a design gets increasingly complex, it may not be as clear cut. For instance, `A` and `B` may be deep hierarchies or rule instances where the lower child is committing a new value of `x`.

This scenario underpins the reasons why the model is transactional, and specifically, why it is locally atomic in its write ordering (e.g. concurrent transactions will choose an order to write – their writes will not overlap). The compiler pro-

vides fast access to potential conflicts, alleviating the difficulties in finding, let alone reproducing, concurrency bugs.

6.2.1 Mutually Concurrent Sets

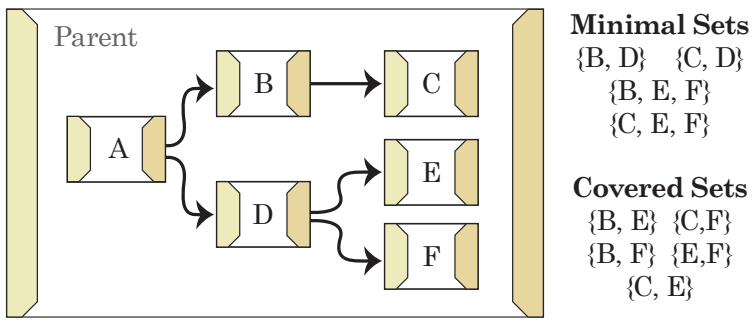


Figure 6.6: Possible concurrent transactions captured in sets. Every covered set is a strict subset of one of the minimal sets. Minimal sets are used for CCR analysis, while the both set types are used in race condition detection (if they commit to the same variables).

A prerequisite for detecting potential race conditions is to determine the sets of potentially overlapping transactions. A straightforward bookkeeping method using *Mutually Concurrent Sets* allows rapid detection in a scalable way. In general, determining these sets on a flat graph would require exponential time; for well-formed hierarchies, however, the runtime is pseudolinear, e.g. the linear component of tree traversal eclipses the local exponential component since the exponent of small numbers is insignificant. Each parent rule manages its overlapping children then compares those sets against the variables they write. If there

Algorithm 6.2 Calculate pairs of potentially mutually concurrent child transactions.

```
1  Precondition: dep_matrix initialized to NoRelation
2  def determineMutualConcurrency(transaction, path):
3    dep_matrix[transaction][transaction] = NoDecision
4    if transaction in path: return
5    foreach t in path:
6     if dep_matrix[transaction][t] != DirectDep:
7       dep_matrix[transaction][t] = IndirectDep
8     if dep_matrix[t][transaction] != DirectDep:
9       dep_matrix[t][transaction] = IndirectDep
10   path.add(transaction)
11   foreach t in forward-dependencies(transaction):
12     dep_matrix[transaction][t] = IndirectDep
13     dep_matrix[t][transaction] = IndirectDep
14     determineMutualConcurrency(t, path)
15   path.remove(transaction)
```

are any transactions writing to the same variable in varying order, the compiler reports the transactions, the variable, and traces the respective assign statements.

6.2.2 Dependency Classifier Matrix

The first step is to traverse the local child dependency graph and construct a dependency classifier matrix. This will indicate, for every pair of children, whether the children have no relationship, a direct dependency, or an indirect dependency. The pseudo-code is listed in alg. 6.2. Once these relationships are identified, it proceeds to set minimal set construction.

Algorithm 6.3 Construct minimal mutually concurrent sets.

```
1  Precondition: dep_matrix is correctly populated
2  def constructConcurrentSets(transaction, dep_matrix):
3    concurrent-sets = {} # set of sets
4    foreach child in transaction.child-rules:
5      foreach other in {transaction.child-rules - child}:
6        if dep_matrix[child][other] == NoRelation:
7          foreach S in concurrent-sets:
8            if other is mutually concurrent with all members of S:
9              S.add(other)
10         if other was not added to any set:
11           concurrent-sets.add({child, other})
```

6.2.3 Mutual Concurrent Set Construction

The dependency classifier matrix identifies relationships between pairs of rules. To minimize the amount of checks required in race conditions (and therefore keep the runtime bounded), a heuristic collects minimal sets of potentially concurrent transactions. The algorithm is listed in alg. 6.3. It maintains a collection of sets that it has constructed so far. For a given transaction's child dependency network, it analyzes the dependency classifier matrix to identify all pairs of child transactions that may run concurrently. For a given transaction, if there exists a set that the transaction is completely mutually concurrent with, then this transaction becomes a member of that set. Otherwise, a new set is formed.

Runtime of this algorithm is N^2M, for N number of child transactions and M mutually concurrent sets. However, as discussed in the next section, the average number of children is low, and thus cubic growth effectively appears as a constant scale factor.

6.2.4 Performance

Limited-child hierarchy is key to effectively linear runtime; Average Number of Transactions per Parent (ANTP) is an approximate metric describing relative system "flatness."

Definition 6.1. *Average Number of Transactions per Parent* (ANTP) is the mean of the number of child nodes for every transaction in the system.

$$ANTP(S) = \frac{1}{N} \sum_{T \in S} C(T_i)$$

where N is the total number of transactions in system S, and $C(T)$ is the number of children in transaction T.

Fig. 6.8 demonstrates effectively linear (*pseudolinear*) runtime for common use of the model – i.e. hierarchies with reasonably low ANTP. In the worst case, runtime is quadratic with the local number of transactions (the immediate child transactions). However, the algorithm uses a path traversal mixed with a set-membership test that scales with path length. Generally speaking, lower ANTP correlates to shorter path lengths – the longest path is at most the number of local nodes.

Random transaction hierarchies of different sizes were generated using a hierarchy construction heuristic. Hierarchies with varying average number of child per parent transaction (ANTP) are binned into ranges $A(0, 50)$, $A(100, 150)$, $A(150, 200)$, and $A(200, 250)$. Note that $A(50, 100)$ yielded similar results to $A(0, 50)$ and is omitted. Fig. 6.7 illustrates the overhead incurred from constructing mutual concurrency graphs with large average child counts. The runtime grows

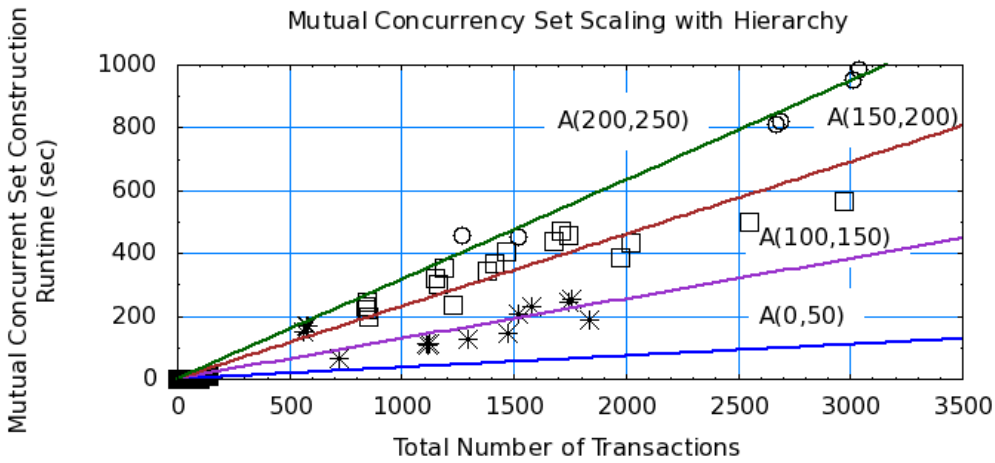


Figure 6.7: Mutual concurrency scaling with different hierarchy depth. "A" indicates the range for ANTP – the average number of transactions per parent. "A(0,50)" means hierarchies with ANTP between 0 and 50.

dramatically as ANTP increases. This is expected – as ANTP increases, the hierarchy disappears, and the algorithm experiences expected close-to-exponential growth.

6.2.5 Isolating and Reporting Race Conditions

As discussed in 3.3.7, this model has a high potential for race conditions. The responsibility of resolving race conditions is left to designers, as they have a better sense of intended behavior. Using mutually concurrent sets, the tool simply performs set intersection operations on the sets of variables written by concurrent transactions.

Since set intersection runs in linear time, race condition detection itself is linear with the total number of transactions (shown in fig. 6.9).

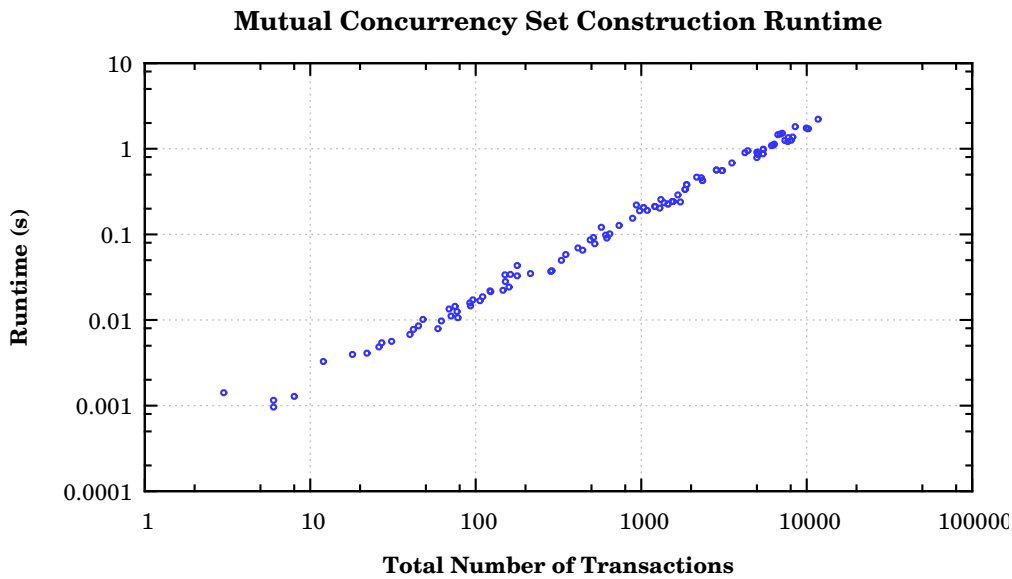


Figure 6.8: Mutual concurrency set construction runtime.

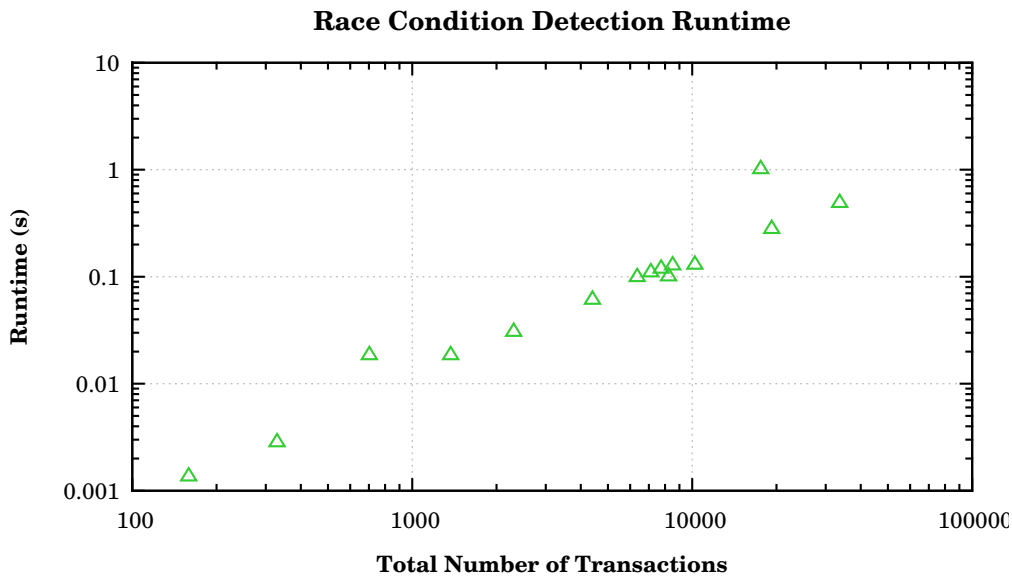


Figure 6.9: Race condition detection scaling.

Algorithm 6.4 Use the dependency classifier matrix to report potential race conditions.

```
for i = 0 to N - 1:
  for j = 0 to N - 1:
    if already checked (i,j): skip iteration
    if dep_matrix[i][j] is not NoRelation:
      A = variables-written(T[i])
      B = variables-written(T[j])
      for variable in set-intersect(A, B):
        report-conflict(variable, T[i], T[j])
```

False Positives The reporting mechanism is conservative – it reports all possible concurrent data interactions. Designers may provide a list of "acknowledged" conflicts, thus avoiding unnecessarily verbose reporting. To further cull the list, the compiler leverages information it gathers from closure mechanism synthesis. The closure logic classifies tokens based on their conditions. If the conditions are mutually exclusive (tagged as such from conditional block structure), then the dependency classifier matrix is updated accordingly.

In the following code, for instance, variable `y` is not reported as conflicting; the compiler statically determines that tokens `tA` and `tB` are mutually exclusive and thus their corresponding forward rules are removed from mutually concurrent sets:

```
rule T:
   if x > 5:
      create tA
   else:
      create tB
rule A(tA):
   y = 1
rule B(tB):
   y = 2
```

6.3 Conversion into Existing Semantic Models

The hierarchical transactional model is a unifying model in the sense that it translates to existing data and execution models in a way that sensible and appropriate for the target domain. In the hardware high-level synthesis field, control/data-flow and pure data-flow graphs are ubiquitous. By translating hierarchical transactions into these models, synthesis tools can leverage a diverse spectrum of synthesis solutions. First, a method of mapping into pure functions is presented (covering straight logic-level synthesis), followed by a control/data-flow conversion technique. Finally, iterator to data-flow conversion is presented (with a novel clustering and synthesis heuristic detailed in Chapter 8).

6.3.1 Transactions to Pure Functions

Data and control mobility means transactions are purely functional – they do not make any changes to state outside of their local context. Therefore, any transaction sub-tree can be converted into a strictly combinatorial and arithmetic expression. In the special case of iterators, they are elaborated and unrolled to form the respective functional nodes.

This process requires an algorithm to select one execution path from the family of possible executions, essentially resolving any concurrent reads/writes by assigning variables a unique name. The general flow is as follows, for rule R_0:

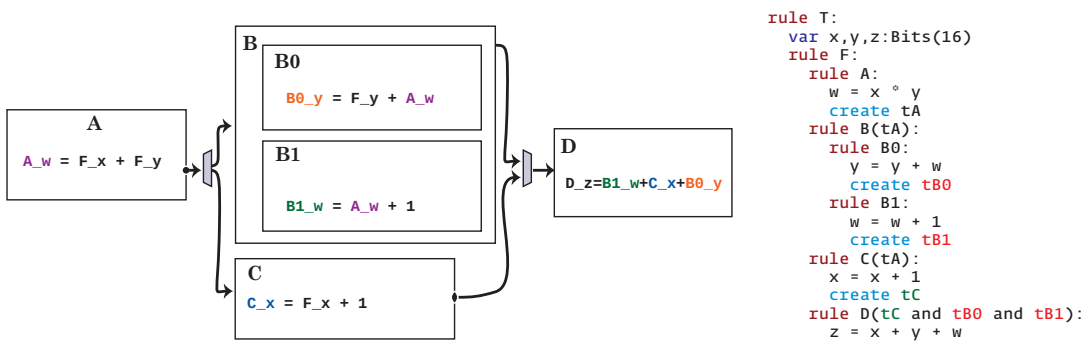


Figure 6.10: Hierarchical transactions to pure function example.

- **For all variables read in by R_0, create source edges**
- **Visit each child rule in topological order**
  - **For every input to the child, resolve the source of information by tracing all of the writes thus far:**
    - **If the variable has not been written by this rule, then its source is the value that was read in by R_0**
  - **For every variable written by the child, create a new edge labeled with the child rule identifier and variable name. Store this information for the next sets of rules**

The process is repeated hierarchically. At lower hierarchies, the algorithm must climb up the local sub-tree to resolve a variable's source, ultimately falling back to the top-level rule's input. For example, if rule R_0 contained child C_0 which itself contained D_0, and D_0 read a variable value that has not been written by either R_0 or C_0, then it would fall back to the value read in by R_0 (from *its* parent).

Fig. 6.10 illustrates one possible execution of the conversion algorithm. It assumes that `F_w`, `F_x`, `F_y`, and `F_z` exist, and represent the values read-in by rule `F`. At each rule it visits, the algorithm traces the source information represented by a given variable. Once each rule has been visited, it performs a final look-up for each variable. If the variable was written internally, the new value overrides the input value. Otherwise, it falls back to the value read-in at the start. It then iteratively substitutes internal values until the final equations are dependent only on the initial inputs read in. The final equations for each of `w`, `x`, `y`, and `z` are presented:

```
1 w: B1_w = A_w + 1
2         = (F_x + F_y) + 1
3 x: C_x = F_x + 1
4 y: B0_y = F_y + A_w
5         = F_y + F_x + F_y
6 z: D_z =                B1_w +     C_x     + B0_y
7         =           (A_w + 1) + (F_x + 1) + (F_y + A_w)
8         = ((F_x + F_y) + 1) + (F_x + 1) + (F_y + (F_x + F_y))
```

Note that each final equations are only dependent on the values read-in.

Once the pure function is constructed, it can be translated directly into RTL. For more complicated pipelining support, sec. 6.4 outlines a method to convert transactions into control/data-flow graphs. The same method addresses iterators, as they are better suited to CDFGs than pure functions.

6.4 Transactions to Control/Data-Flow Graphs

The usefulness of the transaction model is enhanced by its ability to map into many existing approaches. CDFGs and task graphs are the most common application representation model for existing cosynthesis work, and thus reducing

a transaction representation into a data-flow graph grants access to these tools. Translation to control/data-flow graph is very similar to pure function conversion, with the exception that arithmetic nodes are instantiated instead of functions. In fact, pure functions and CDFGs are isomorphic representations of the same behavior. The primary benefit with direct CDFG representation is the availability of scheduling tools and their ability to convert iterators efficiently. While it is possible to create pure functions from iterators, it would lead to very large functions.

One key point here is that transaction data copying allows this process to occur without requiring arbitration or data coherence management. It simply selects a subset of the valid executions and replaces copying by relabeling state. This procedure removes the copying semantic; however, it should be noted that the existence of encapsulated state fundamentally allows an easy and straightforward conversion. The same applies to hierarchy – at any scale, the heuristic can flatten the hierarchy into a graph. This allows the synthesis tool to determine the appropriate granularity for conversion.

Consider the following example:

```
rule Top<-(w, x, y, z):
var m, n, p, q: Bits(16)
   rule A:
      m = x * y
      p = w * z
      create tA
   rule B(tA):
      rule B0:
         n = p + x
         create tB0
      rule B1(tB0):
         q = m + y
         if m > n:
            create tB1
   rule C(tA):
      p = m + x
    rule D(tB1):
       q = q + 1
```

As is the procedure with pure functions, the first step is to rename variables according to their source. All of the inputs and outputs are collected, and edges are generated for each.

1. First step is to rewrite all of the variable names and assign nodes – at this stage, the tool will effectively select a subset of possible executions. By naming variables, it selects the source rule, even if another rule may write that value. For instance, rule `C` uses `A`'s copy of `m`.

2. Rewrite each rule's ID. Create edges for every variable read-in and written out.

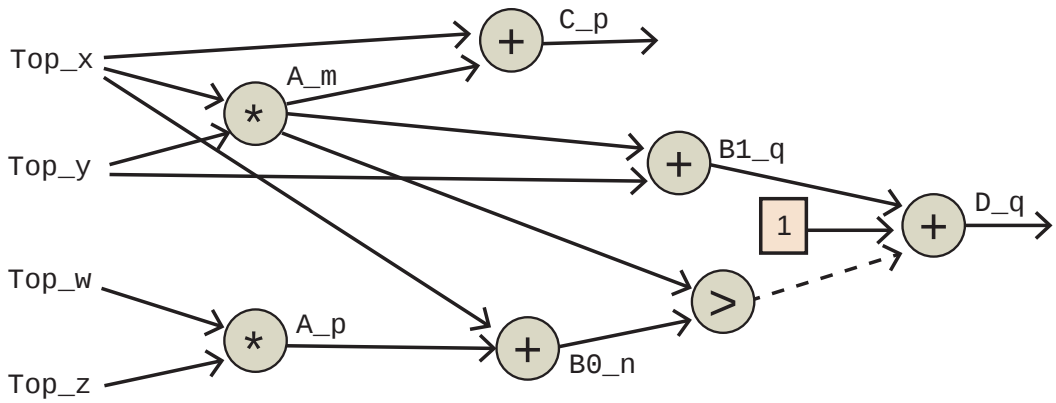


Figure 6.11: Hierarchical Transactions to CDFG Example. Solid lines are data edges. Dotted lines are control edges.

The resulting example becomes:

```
1  rule Top<-(Top_w, Top_x, Top_y, Top_z):
2     rule A:
3        A_m = Top_x * Top_y [node 0]
4        A_p = Top_w * Top_z [node 1]
5        create tA
6     rule B(tA):
7        rule B0:
8           B0_n = A_p + Top_x [node 2]
9           create tB0
10       rule B1(tB0):
11          B1_q = A_m + Top_y [node 3]
12          if A_m > B0_n: [control node 4]
13             create tB1
14    rule C(tA):
15       C_p = A_m + Top_x [node 5]
16    rule D(tB1):
17       D_q = B1_q + 1 [node 6]
```

Fig. 6.11 illustrates the result of creating a CDFG from the above example – solid lines represent operands passing between operations while dotted edges are purely control (and conditionally determine flow). The token `tB1` is created conditionally based on the difference between `m` and `n` – a conditional edge is created between the comparison operator and the subsequent addition.

6.4.1 Flow-graphs from Iterators

While conversion from iterators to CDFGs makes the representation amenable to existing technologies, it does so at the cost of lost symmetry information. There is high potential for leveraging symmetry in code generation as a compaction method for reducing control overhead, particularly when synthesizing addressing/indexing logic. For example, a system with an accelerating arithmetic pipeline would need input and output addresses for every operand, along with a method to represent data dependencies. With symmetry information, this control information can be codified into a state machine index generator optimized for a particular architectural target.

That said, there are still benefits to sourcing CDFGs from iterators. First and foremost, iterators make guarantees about the scope and access of information. They will not result in random graphs, but rather, well-structured, well-defined graphs.

The actual translation from transactions containing iterators into CDFGs is precisely the algorithm above, with an added iterator unrolling step. The system first converts the body of the iterator; it constructs a local CDFG and replicates this set for each iteration. For every iterating variable, the compiler queries its respective iterators (e.g. if they are composed or nested iterators) to yield the requested data.

6.4.2 Software Scheduling

Once a portion of the application is targeted at the software domain, the synthesis tool can selectively flatten subsets of the design and apply traditional, well-known software scheduling techniques. In a multiprocessing context, transactional boundaries resolve the problem of managing shared state – all of the interaction is well-defined, and potential race conditions are exposed at the abstract model level. There are software-specific features that can guide efficient realization: a large, cached memory is useful for storing dependency information for dynamically dispatching transaction executions. The software synthesizer may leverage reference locality to better use available caching. If the synthesis tool is aware of cache sizing and structure, it can accordingly adjust software/hardware partitioning to make (near-)optimal utility of the cache.

Chapter 7

Control-Dominant Application Study

Real-world hardware/software systems exhibit a mix of control- and data-dominated components. This chapter demonstrates the control end of the spectrum: a study into the specification of a microprocessor, complete with internal concurrent tasks and complex control decisions. The emphasis is on exercising the language and semantic model – in practice, a processor may be suitable as a cosynthesis application for simulation speed-up. For instance, in the early stages of a microprocessor instruction architecture design, a field-programmable gate array (FPGA) can leverage hardware accelerator to rapidly simulate applications, exposing potential pipeline bottlenecks and critical instructions.

More importantly, if the language can express a processor, then it is rich enough to express any equally complex application (including the type of mixed reactive/high performance applications commonly targeted towards embedded systems). The primary goals of this study is to evaluate feasible execution of

a complex design expressed as hierarchical transactions, and expose useful properties from an exercised model.

7.1 The MSP430 Microprocessor

Texas Instrument's MSP430 [81] is a popular ultra low power microprocessor commonly used for power-constrained embedded applications. Its instruction set architecture (ISA) is minimal but complete. The HTL realization is based on Greg Hoover's PyTDL MSP430 implementation [48], extended with hierarchy and covering the complete ISA.

7.2 Processor Specification

When approaching this problem, the primary focus is on expressing the interface-level functionality, completely irrespective of timing details. The design is an abstracted implementation of a pipelined architecture – functionally correct, but latency tolerant. It runs valid MSP430 executables (generated from a GCC-based cross compiler).

The instruction set contains standard addition and subtraction operators – a multiplier is accessible through memory mapped I/O. Decoding and executing instructions are expressed directly in rules. Both the register file (containing 16 registers), and memory are instances of the `Addressable` built-in library type, allowing indexed behavior. The topmost rule is a testbench environment, containing the entire application's behavior within a single rule tree. Instructions

and status registers are object instances of specialized classes – the `Instruction` class contains functional rules for decoding the operation. The results in clean, readable code.

The following sections demonstrate a small portion of the specification (the decoding logic). The full specification is available in Appendix B.2.

7.2.1 Decoding

The following decodes an instruction, creating appropriate conditional tokens depending on the type of operation.

```
var src_op, dest_op: Bits(16)
var src_pc_offset, dest_pc_offset: Bits(2)

var src, dest: Bits(4)
var smode, dmode: Bits(2)

rule Decode:
   rule DetermineType:
      print "Instruction value: ", hex(instr.v)
      pc_offset = 1
      if instr.is_jump():
         print "  Jump operation"
         create tExecuteJump
      elif instr.is_single():
         print "  Single operation"
         create tDecodeSingle
      elif instr.is_double():
         print "  Double operation"
         create tDecodeDouble
      else:
         create tDecodeError

   rule DecodeSingle(tDecodeSingle):
      src = instr.single_dsreg()
      smode = instr.single_ad()

      if not instr.opRETI():
         create tGetSourceOperand
      else:
         # For RETI instructions, we just execute them
            directly. We just pretend
```

```
31          # that the source operation was completed by
                creating the token here.
32          src_op = 0
33          src_pc_offset = 1
34          create tSourceReady
35
36      rule DecodeDouble(tDecodeDouble):
37          src = instr.double_sreg()
38          smode = instr.double_as()
39          create tGetSourceOperand
40
41          if instr.opMOV():
42              create tDecodeMOV
43          else:
44              dest = instr.double_dreg()
45              dmode = instr.double_ad()
46              create tGetDestOperand
47
48      rule DecodeMOV(tDecodeMOV):
49          # If it's a MOV instruction, we do not need to read
                the destination
50          # so we just create the token here.
51          if smode == 1:
52              dest_pc_offset = 1
53          else:
54              dest_pc_offset = 0
55          dest_op = 0
56          create tDestReady
57
58  # end rule Decode
```

Note that the print statements are purely for simulation. Recall that all rules are atomic – in the case that there are multiple statements in the rule, they run concurrently.

The `DecodeSingle` and `DecodeDouble` paths are mutually exclusive, conditional on the type of instruction. The token semantic lends itself to a flexible method of communicating control that particularly simplifies error conditions. On decode error, the `tDecodeError` token is created – the token climbs the hierarchy (as the respective transactions commit) until it is consumed by logic to manage the decode error.

The processor dispatches concurrent paths to retrieve its source operands (including the constant generator, memory, or register). It further sets up the destination operand. Execution is distributed: jump instructions, arithmetic operations, and memory read/writes are all rule sub-trees. Finally, write-back occurs after execution, ending the instruction stream.

7.3 Race Condition Detection

The race condition detection algorithm from sec. 6.2.5 was applied to the processor. Since the heuristic is conservative, it reported over 40 situations where variables may be overwritten concurrently. The list was culled based on designer knowledge of exclusive execution paths down to eight potential issues. After analyzing the reads and writes, two serious race conditions were exposed, both related to off-by-one counting in the calculation of the program counter. The entire process of detecting and correcting race conditions took less than five minutes – there was no additional overhead in specifying a separate formal model or requiring deep execution analysis. The direct codification of shared state allowed the compiler to quickly and efficiently report these bugs to the designer.

7.3.1 The Original Problem

In the first version of the processor specification a `pc_offset` variable indicated how much to increment the program counter. Initially, no concurrency was exploited, and the rules to retrieve source operands each incremented this off-

set. Once the two paths were made concurrent, however, it exposed the following conflict:

```
var pc_offset: Bits(2)

rule GetSourceOperand(tGetSourceOperand):
   rule DetermineSourceAddressing:
      pc_offset = 0
   rule SourceAbsolute(tSourceIsAbsolute):
      pc_offset = pc_offset + 1
   rule SourceIndexed(tSourceIsIndexed):
      pc_offset = pc_offset + 1
   rule SourceIndirect(tSourceIsIndirect):
      pc_offset = pc_offset + 1
   rule SourceImmediate(tSourceIsImmediate):
      pc_offset = pc_offset + 1
   # Eventually creates tDestReady

rule GetDestOperand(tGetDestOperand):
   rule DetermineDestAddressing:
      pc_offset = 0
   rule DestAbsolute(tDestIsAbsolute):
      pc_offset = pc_offset + 1
   rule DestIndexed(tDestIsIndexed):
      pc_offset = pc_offset + 1
   rule DestIndirect(tDestIsIndirect):
      pc_offset = pc_offset + 1
   rule DestImmediate(tDestIsImmediate):
      pc_offset = pc_offset + 1
   # Eventually creates tSourceReady

# an intermediate rule waits for tSourceReady and tDestReady
# and creates tInstructionDone

rule InstructionDone(tInstructionDone):
   pc = pc + pc_offset # Here's where the two paths report the
        wrong PC offset
```

Once the problem was identified, the solution was simple:

```
rule GetSourceOperand(tGetSourceOperand):
   rule DetermineSourceAddressing:
      src_pc_offset = 0
   rule SourceAbsolute(tSourceIsAbsolute):
      src_pc_offset = src_pc_offset + 1
   rule SourceIndexed(tSourceIsIndexed):
      src_pc_offset = src_pc_offset + 1
   rule SourceIndirect(tSourceIsIndirect):
      src_pc_offset = src_pc_offset + 1
   rule SourceImmediate(tSourceIsImmediate):
      src_pc_offset = src_pc_offset + 1
   # Eventually creates tSourceReady

rule GetDestOperand(tGetDestOperand):
   rule DetermineDestAddressing:
      dest_pc_offset = 0
   rule DestAbsolute(tDestIsAbsolute):
      dest_pc_offset = dest_pc_offset + 1
   rule DestIndexed(tDestIsIndexed):
      dest_pc_offset = dest_pc_offset + 1
   rule DestIndirect(tDestIsIndirect):
      dest_pc_offset = dest_pc_offset + 1
   rule DestImmediate(tDestIsImmediate):
      dest_pc_offset = dest_pc_offset + 1
   # Eventually creates tDestReady

rule Execute(tSourceReady and tDestReady):

   rule DetermineExecutionType:
      # first combine pc offsets
      pc_offset = pc_offset + dest_pc_offset + src_pc_offset
```

Generalized Model The primary limitation in race condition detection is indexed data, particularly access to the register file and memory. These situations encounter traditional exponential scaling, as there are few options except comprehensive state exploration. While the specific violating indexes cannot be ascertained at compile time, it does report when transactions attempt to concurrent write to the indexed object. In the worst case, it provides overly conservative bounds.

7.4 Transactional Simulator

The MSP430 executed in the behavioral simulator, validating that the architecture was correctly implemented. Simple software applications were written in C and cross-compiled to the MSP430 ISA. These applications were then executed in the transactional simulator (built into the compiler), preserving the token and transactional abstraction. Development time was short, largely because the specification language is concise and clear.

In order to facilitate the register file and memory, an extension was built into the simulator that prevented it from copying either indexable object transactionally. This would have created tremendous memory and performance overheads, hindering simulation speed. Instead, a logging mechanism maintained a trace of reads/writes to the parent memory (trimming overwritten information to keep it small). Any concurrently executing transactions were resolved at commit; if there was a memory conflict between transactions, this information was dynamically reported to the user.

The following example illustrates a situation that the simulator reports. For rule `A`, the logging mechanism stores a mapping between `memory[1]` and its write value `1`. The same happens for rule `B` – the value `2` is stored for `memory[1]`. Once both transactions commit, the simulator detects a conflict and warns the user.

```
1 rule Top:
2   var memory: Addressable(1024, Bits(16))
3   rule A:
4     rule A0:
5       x = 1
6     rule A1(tA0):
7       memory[x] = 1
8   rule B:
9     memory[1] = 2
```

7.5 Direct-to-RTL Realization

Designing new semantic models requires a mix of formal construction and practical realization. Early in the design process, a direct-to-RTL code generator was used to discover any potential execution issues – in fact, this led to the discovery of a closure requirement. The code generator directly implemented transactions in the Verilog Hardware Description Language, including full copying semantic. For the register file and memory, however, the code generator instantiated single instances of both and scheduled around concurrent accesses (for simplicity). While this approach obviates any race conditions related to either indexed storage, it does allow full simulation of the control logic.

The primary insight in direct model realization is verification of closure correctness. The MSP430 implementation contains a significant number of conditional tokens, and therefore conditionally executing transactions. In the gate-level simulation, a compiled C program executed on the RTL MSP430 implementation, covering a reasonable array of instructions. By virtue of complete execution, the model proved to be closed and executable.

Chapter 8

High Performance Arithmetic Applications

At the opposite end of the spectrum from control-dominated applications are high-performance, arithmetic-heavy algorithms. This chapter demonstrates a feasible synthesis path for high-performance realizations of arithmetic-heavy application. It presents architectural exploration of matrix multiply, Fast Fourier Transform (FFT), and image convolution examples. The assumption is that lower level model conversion occurs after initial coarse-grained synthesis – i.e. after information flow is grossly organized with a global view of the problem. It leverages encapsulated control and state to reduce a larger problem into a format that has well-known and field-tested solutions without the need to mitigate data coherence at a large scale. The following sections expand upon data-flow methods to include a unique memory capacity-based clustering technique ideal for hardware arithmetic acceleration. This approach scales tremendously, managing graphs up to the million-node scale.

8.1 Memory Capacity and Data Motion

Transactional semantics accomplish dual goals of pseudo-linear scalable analysis without sacrificing performance. The latter arises from a decoupling of specification and hard memory constraints. As a result, synthesis has freedom to allocate and bind to complex memory systems.

Capturing data motion costs is critical to efficient heterogeneous implementations – it motivates the design of the hierarchical copying semantic. When approaching synthesis of a very large application, memory capacity constraints will significantly dictate the resultant performance. If there is not enough intermediate, local storage to sufficiently provide operands to arithmetic pipelines, then a scheduler has little choice but to insert bubbles, each of which is a missed opportunity.

Hierarchical transactions allow synthesis tools to easily summarize and isolate subsections of a design without the need for complex and computationally expensive clustering. In fact, clustering a directed graph (say, for data-flow representations) for a very large problem can be difficult: the problem is naturally tightly constrained, as the clusters need to remain acyclic to remain amenable to scheduling.

8.2 Problem Setup

The primary target of the high-performance data-flow technique presented is reconfigurable embedded systems containing a microprocessor (either in the fabric or a hard IP block) interfacing with a large, off-chip double data rate

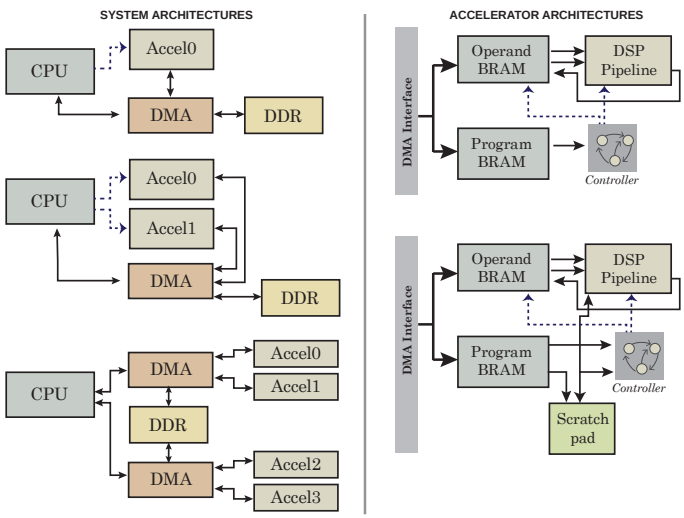


Figure 8.1: Left: System point-to-point architectures with 1, 2, and 4 accelerators. Right: Accelerator without and with scratchpad memory (SPM)

(DDR) memory. The processor runs software that dispatches operand clusters to accelerators through a direct memory access (DMA) controller – that is, the software instructs the DMA to transfer DDR memory directly to and from the accelerators. All of the buses are assumed to be point-to-point to reflect modern ARM-based platforms. Fig. 8.1 illustrates three system architectures and two accelerator architectures.

8.2.1 Commercial IP Options

There are a variety of off-the-shelf third party signal processing accelerators for algorithms such as DCT, FFT, and matrix multiply, many of which contain bus interfaces to allow simple software integration. These products are remarkably efficient but are typically targeted as fixed sizes. Applications like FFT and DCT can handle larger problems by composing smaller accelerators (e.g. a $16K$-point

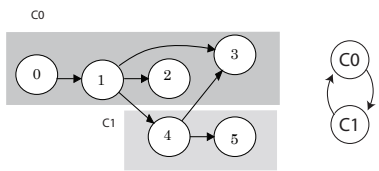


Figure 8.2: Illustration of clustering that can induce cycles

FFT can be composed of $4 \times 8K$ FFT blocks). However, as the problem grows to very large scales, there are no obvious methods of managing the enormous amount of information that must be exchanged between blocks. Algorithms such as matrix multiply have no choice but to resort to DSPs and custom DSP software. Again, managing the gross motion of data is not straightforward. The approach presented in this chapter accepts designs of any shape. While random graphs may prove less efficient, real-world arithmetic graphs can exploit hardware parallelism due to their local operation scope.

8.2.2 Challenges in Digraph Clustering

Clustering a directed graph can lead to a minefield of induced cycles (fig. 8.2) when the cluster graph is generated based on its nodes dependencies. This cannot be directly scheduled as there are coupled dependencies which are unresolvable without violating cluster boundaries. The Kernighan-Lin algorithm [54] approaches this problem with a heuristic that evaluates the net cost of preserving balanced partitions across multiples moves from one partition to another, capturing both the cost of the node itself and the subsequent corrections that make the original move sensible.

Scheduler-driven partitioning proposes a different approach: clustering is performed as part of list scheduling, thus guaranteeing that the clusters themselves can be scheduled at a coarser grain.

8.3 Scheduler-Driven Partitioning

To reach high performance arithmetic designs, an implementation strategy is presented based on well-formed data-flow designs generated by iterators. The target platform is a heterogeneous reconfigurable architecture. The primary application space is reconfigurable embedded systems, where there is an opportunity to selectively hardware-accelerate an arithmetic-heavy portion of the application. Its approach partitions a data-flow graph into small clusters that fit into microarchitectures consisting of a program memory, an operand memory, and a pipelined micro-DSP (i.e. a DSP without any control logic), shown in fig. 8.3. The system architecture contains a primary CPU attached to high-speed, high-capacity double-data rate (DDR) memory, along with a selection of accelerators (fig. 8.1). The peripheral has an optional scratchpad memory (SPM) with parameterizable size. A scratchpad memory essentially provides a local operand buffer to exploit reference locality without the overhead of a traditional cache. In essence, it provides a second pair of memory ports to allow maximal use of every computation cycle available.[74]

The first step selects the arithmetic-heavy portion of the application, and constructs a data-flow graph via its iterators. A list scheduler then selects nodes to insert into the DSP pipeline until memory capacity constraints are met, thus

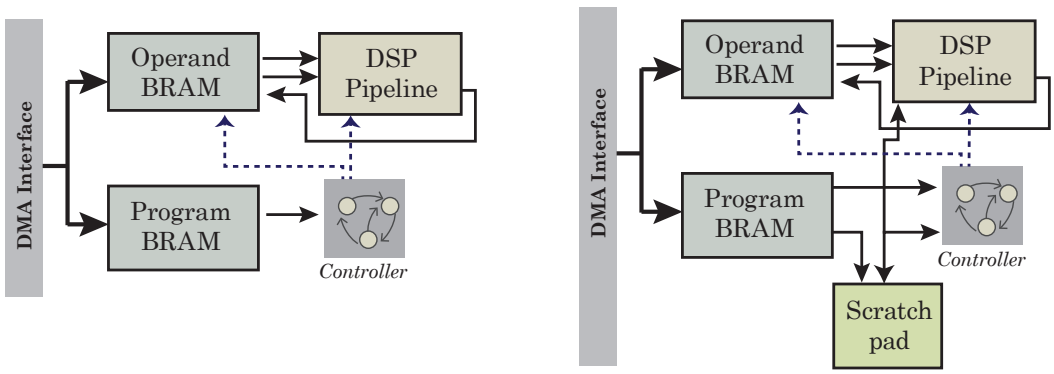


Figure 8.3: Internal architecture for peripheral. The bus provides initial operands and the program. A controller executes this program and uses the local operand block RAM (BRAM) to store intermediate operations. After the program has completed, the controller initiates a transfer to the DMA controller while simultaneously asserting an interrupt for the main CPU. When the DMA is configured in scatter-gather mode, the next set of operations arrives automatically, based on an internal linked list of data copying blocks. The peripheral on the right adds a local scratchpad memory to provide local, higher utility access to operand data.

Algorithm 8.1 List-scheduler based cluster building heuristic

```
1: function BuildCluster(graph, ready-list, bram-size)
2:     populate ready-list with initial nodes from graph
3:     alloc-mgr ← AllocationManager()
4:     while graph_{not-scheduled} do
5:         alloc-space ← alloc-mgr.space()
6:         if alloc-space at capacity and ready-list is locked then
7:             exit loop
8:         end if
9:         new-node ← ready-list.dequeue(alloc-space)
10:        if new-node is valid then
11:            schedule(new-node)
12:            add fwd deps to ready-list
13:            allocate addresses for new-node
14:        else
15:            wait for pipeline to clear
16:        end if
17:    end while
18:    return schedule, nodes-scheduled
19: end function
```

delineating that particular cluster. The global clustering heuristic creates a corresponding cluster node, then continues generating further clusters until all of the nodes have been scheduled and clustered. The ready list for the scheduler must be persistent across different cluster calls, as it maintains the overall scheduler state. A specialized allocation manager assigns memory addresses to the input and output operands as they are available. If there is no operand space remaining, the scheduler first determines if it can wait until the pipeline has cleared. If there are no more operations that can fit within the constraint, it terminates the local run. Pseudocode for the clustering-scale heuristic is shown in alg. 8.1.

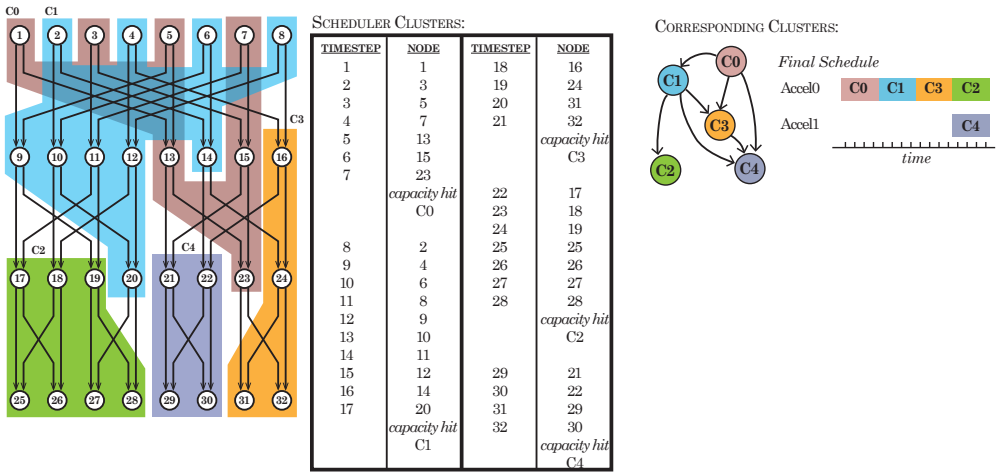
Algorithm 8.2 Design partitioner using the cluster building heuristic

```
1: function ClusterGraph(graph, bram-size, bram-count)
2:     compute node priorities
3:     ready-list ← ReadyList()
4:     populate ready-list with initial nodes
5:     while ready-list has nodes do
6:         cluster-schedule, nodes-scheduled ← BuildCluster(graph, ready-list, bram-size)
7:         clusters.add(new Cluster(nodes-scheduled))
8:     end while
9:     cluster-graph ← BuildClusterGraph(clusters)
10:    cluster-schedule ← ResourceSchedule(cluster-graph, bram-count)
11: end function
```

Once the initial clusters are constructed, the clusters themselves are scheduled to determine total system cost. It factors in bus timing to transfer information to and from the accelerators and accepts a parameter specifying the number of available peripherals. A simple resource-constrained list scheduler is used (since the total number of clusters tends to be relatively small).

8.3.1 8-point FFT Clustering Illustration

Consider the scheduler-driven clustering of a small 8-point FFT. Fig. 8.4 illustrates the first phase of the heuristic that gathers individual clusters based on capacity. The image highlights five clusters, and shows the order in which they may be scheduled on the right (one of many valid orderings). In practice, the heuristic will select much larger and deeper clusters; for illustrative purposes, assume a very small capacity constraint. Once the clusters are generated, the cluster dependencies are constructed and the graph is scheduled onto the system. Fig. 8.4 illustrates the resultant cluster graph (based on the FFT's dependencies) and



TIMESTEP	NODE	TIMESTEP	NODE
1	1	18	16
2	3	19	24
3	5	20	31
4	7	21	32
5	13		*capacity hit*
6	15		C3
7	23		
	capacity hit	22	17
	C0	23	18
		24	19
8	2	25	25
9	4	26	26
10	6	27	27
11	8	28	28
12	9		*capacity hit*
13	10		C2
14	11		
15	12	29	21
16	14	30	22
17	20	31	29
	capacity hit	32	30
	C1		*capacity hit*
			C4

Figure 8.4: Scheduler-driven clustering of FFT with corresponding cluster graph and system-level schedule for FFT

a potential schedule onto an architecture with two accelerators. In this particular example, the second accelerator can only be used 25% of the time; this valuable insight can guide the designer to select the appropriate number of accelerators based on their constraints.

8.3.2 Target Algorithms

Fast Fourier Transform The Fast Fourier Transform (FFT)[20] is a common signal processing algorithm that transforms a time-varying signal into its frequency components (along the power spectrum). It is a prime candidate for hardware acceleration, as it exhibits a high ratio between computations and data motion. This shape also models similar algorithms such as the Discrete Cosine Transform (DCT) commonly used in multimedia applications.

Matrix Multiplication Matrix multiply is a foundational operation underpinning signal processing and control system applications, not to mention graphics, learning, and physical simulation tools. Matrix multiplication has been implemented for any valid matrix shapes, allowing a large variety of linear operations (incidentally including a variant of FFT[32]). The iterator produces a tree reduction for the final sums when computing the linear product of rows and columns.

Image Convolution Finally, an image convolution was chained together with both a matrix multiply and FFT to emulate a complex DSP algorithm. This configuration illustrates that the heuristic is agnostic of problem shape, allowing a wide variety of composed iterators.

Software Pipelining Software pipelining is the dispatch of independent threads of execution. Issuing multiple independent computational threads (for instance, on different video frames) allows maximal use of hardware parallelism. Computations can overlap with bus transactions, and independent operations fill in otherwise wasted computation cycles. Implementing this in the scheduler involves replicating the cluster graph, then scheduling all of the separate cluster graph copies onto the same system architecture.

8.4 Results

This heuristic was applied to a series of FFTs, matrix multiplications, image convolution, and combinations of these to emulate DSP pipelines. The parameters dictating algorithm size (for instance, FFT width, matrix dimensions) were varied to evaluate scaling. The clustering and scheduling heuristics were implemented in Python. Verilog models were written for the accelerating peripheral, including the bus interface logic. The bus cost models reflected a Xilinx Vivado system using DMA Scatter-Gather targeting a Zedboard[90]. All results were generated on a dual-Xeon E5630 server. A tremendous number of valuable architectural configurations were generated automatically – within a single day, over 500 architecture configurations were evaluated, varying BRAM size, SPM size, algorithms, number of accelerators, and software pipeline depth.

8.4.1 FFT

The first demonstration is the Fast Fourier Transform (FFT). The scheduler-driven clustering heuristic was applied to FFTs of varying sizes across different local memories. The FFT results are listed in appendix D, listing for given BRAM/accelerator configurations, how long the resultant schedule is, the number of clusters generated, and the heuristic runtime. All results were run on a second-generation 16-thread Intel Core i7 server.

Fig. 8.5 illustrates, for different FFT sizes, the length of the generated schedule versus the size of the accelerator's BRAM for two accelerators. It is clear that in all cases, there was improvement with added memory capacity; that improvement,

however, is limited (on the order of 6-8%). This is due in large part due to limited memory port access.

Unexpected BRAM-size Invariance One of the key insights discovered by this heuristic is the ineffectiveness of larger operand BRAMs. Fig. 8.8 illustrates, for varying BRAM sizes, the effect of scratchpad memory size. For reasonably sized elements (enough to cover a saturated DSP pipeline), the system executes with relatively high utility of the arithmetic units. This is enforced by the dense arithmetic shape of FFT and matrix multiplication which always provide ready operands.

Without understanding how the system actually executes, this result is rather unintuitive. Larger local memories are assumed to correlate with higher performance. Clearly, the specific architectural choices in these accelerators exhibit unexpected cost trade-offs. By rapidly exposing the key architectural effects, the designer can tailor systems with much lower area (due to smaller BRAMs), allowing better use of the remaining programmable fabric.

Importance of Scratchpad Size On the other hand, the scratchpad size has a significant effect on performance. Despite a relatively small size, the additional memory ports prove invaluable to ensuring there are new operands at almost every clock cycle. It is safe to conclude that efficient performance for this style of accelerator requires a balance between memory capacity and, perhaps more importantly, memory port resources.

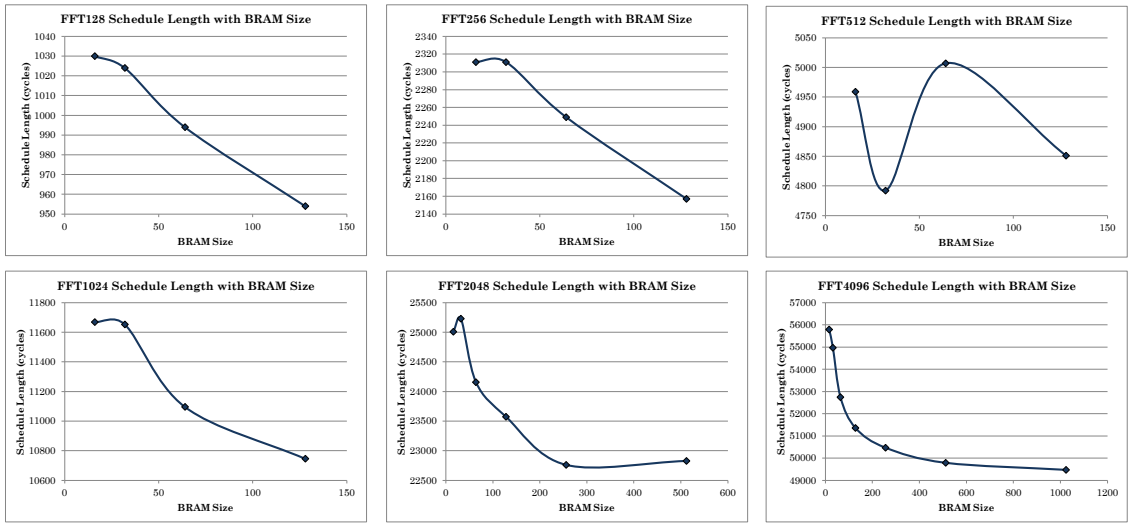


Figure 8.5: FFT total schedule length (including bus transfers) versus size of BRAM for a 2-accelerator architecture.

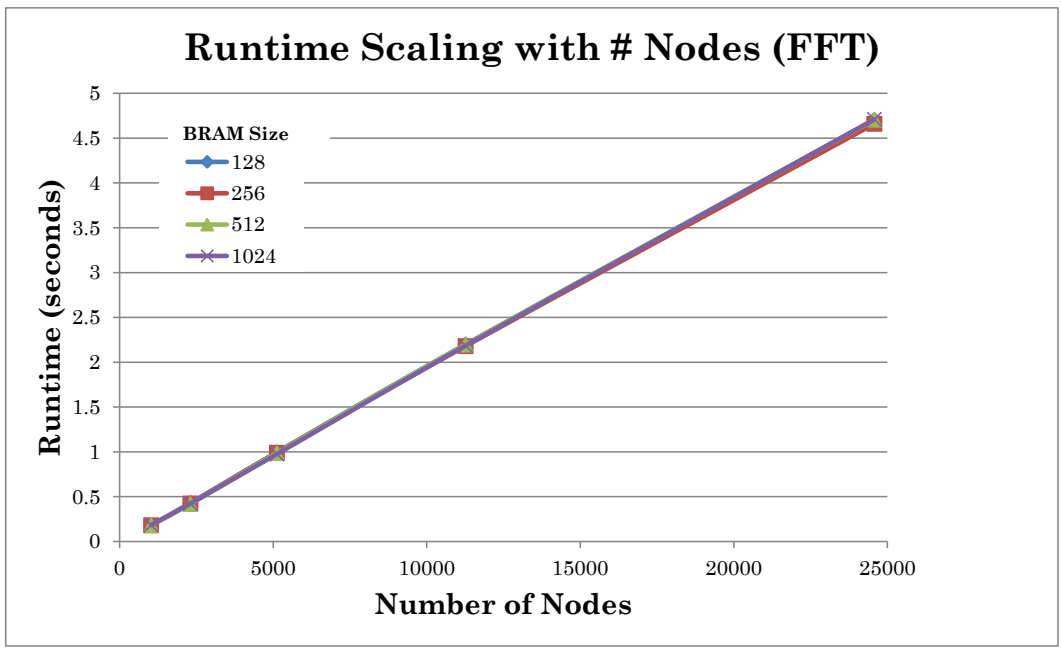


Figure 8.6: Heuristic runtime scaling with increasing node counts on FFT instances.

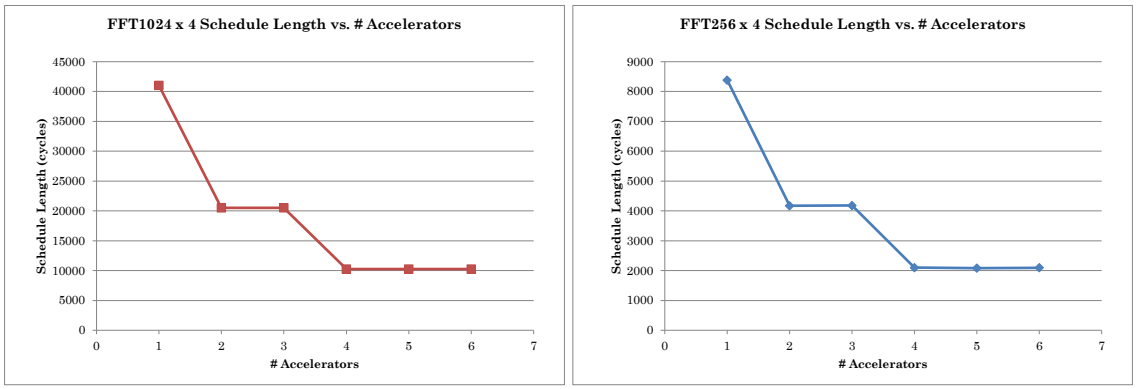


Figure 8.7: For fixed BRAM size, both FFTs of size 1K and 256 exhibited the same shape across different peripheral accelerator configurations.

Fig. 8.6 demonstrates the heuristics scaling with the number of nodes, across different BRAM constraint sizes. As is evident, scaling is linear as the number of nodes increases, irrespective of BRAM size.

8.4.2 Matrix Multiplication

The same experiments were run on matrix multiplication – square matrices were multiplied together, across varying matrix sizes and BRAM constraints. Fig. 8.9 demonstrates nearly identical behavior as the FFT – as the BRAM size constraint increases, there is diminishing parallelism due to increased program length.

Software-pipelined Matrix Multiplication A matrix multiplication was software pipelined against multiple accelerators. The intuition is that the number of accelerators should correlate to the depth of the software pipelining. Indeed, fig. 8.10 shows this precise behavior. Interestingly enough, for pipeline depths of 7

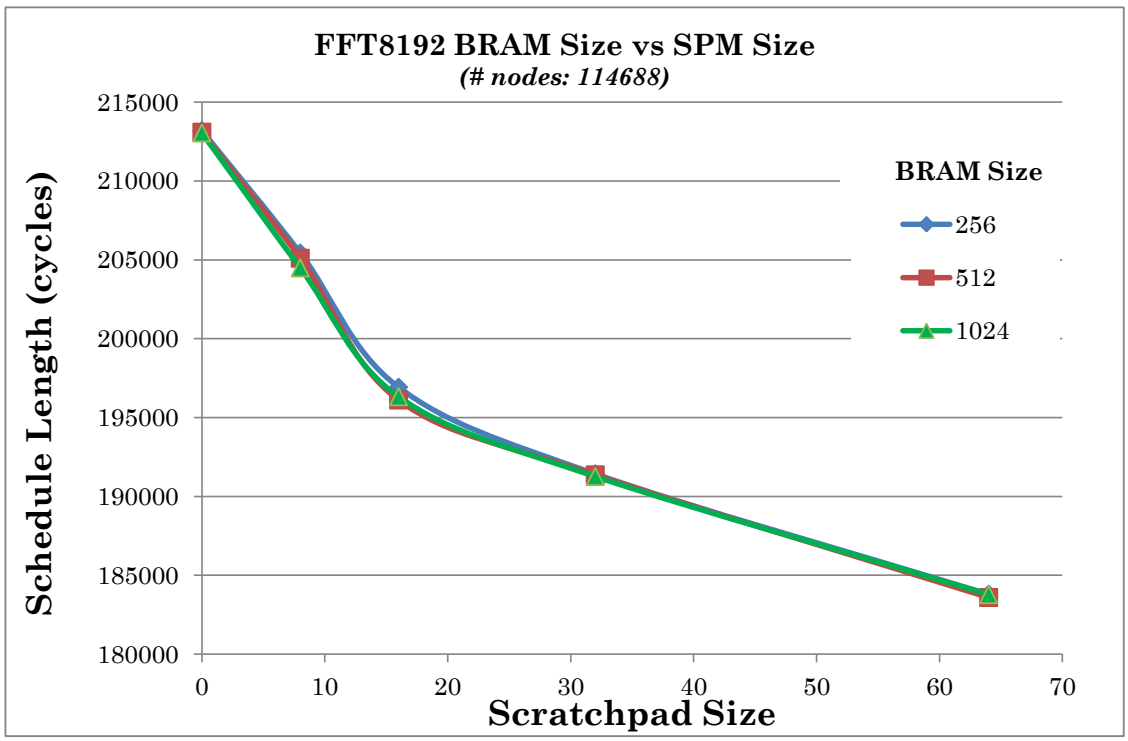


Figure 8.8: Across different BRAM sizes, the SPM size was varied. Clearly, there is little to no improvement with larger BRAM. This correlates to utility limits stemming from a single port constraint.

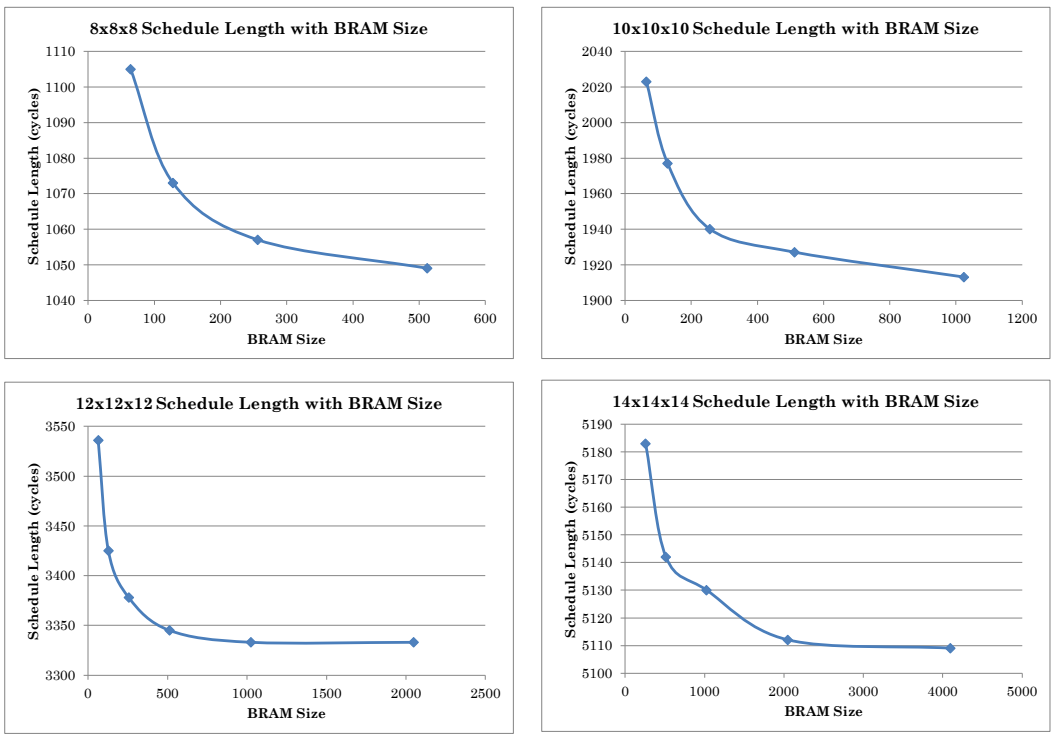


Figure 8.9: Matrix multiply total schedule length versus BRAM size constraint for 2-accelerator architecture.

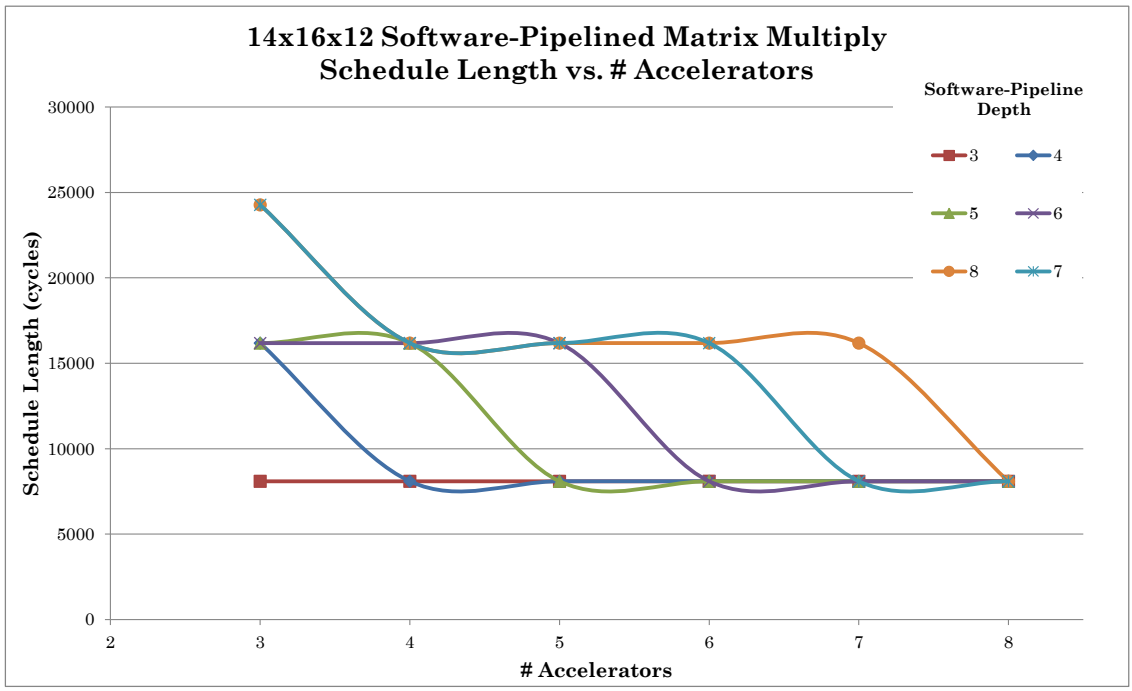


Figure 8.10: Matrix multiplication (of a 14x16 matrix with a 16x12 matrix) performance versus software pipeline depth & number of accelerators. It is interesting to note the trade-off between performance and device area. The jump for high-depth software pipelined designs shows a Pareto point that trades throughput for area.

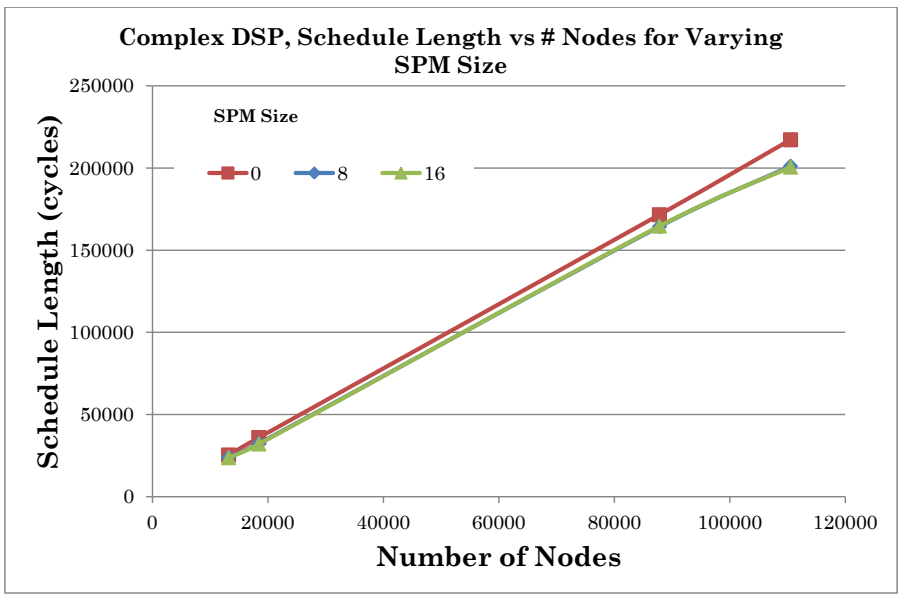


Figure 8.11: Scaling for complex DSP application illustrates the heuristics ability to preserves near-linear scalability for a variety of design shapes.

and 8, the gap between 3 and 4 accelerators is significant. This configuration may prove to be a better area/performance trade-off.

8.4.2.1 Complex DSP Algorithm

The final demonstration chains image convolution with matrix multiply and FFT, shown in fig. 8.12. All three algorithms form a single graph. Fig. 8.11 shows a comparison of schedule length versus problem size for different scratchpad sizes. Interestingly, as the problem size scales up, the schedule length scales linearly, reflecting the tightly knit nature of the input data-flow graph.

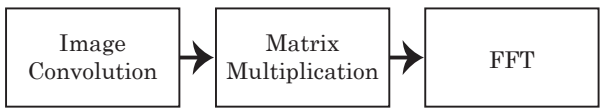


Figure 8.12: Algorithms chained together to form a complex DSP demonstration.

8.4.3 Utility and BRAM vs SPM Effectiveness

The primary performance differentiator is the scratchpad memory size. To evaluate why it eclipses BRAM size, fig. 8.13 demonstrates, for each application, the ratio of the cycles in which the DSP was occupied versus unoccupied/idle (the *utility*). For varying size of BRAM, there is little difference in utility. However, when the scratchpad memory is introduced, the additional memory ports allow better utility of computational cycles, and therefore yield faster schedules.

Fig. 8.14 shows the scratchpad utility over time (fig. 8.14a) and correspondingly the arithmetic utility over time (fig. 8.14b) for selected clusters of an FFT schedule. The values are cumulative – where it increases, it has made use of the resource. When it flattens out, it is waiting for the pipeline to clear or for memory ports to free. The majority of clusters exhibit this shape, as the algorithms are relatively dense and always have available operation.

8.5 Scaling to Very Large Problems

The final demonstration chains increasingly larger FFTs to matrix multiplication outputs. These were then chained together in long, concurrent pipes, forming a single graph into the millions of nodes. Fig. 8.15 shows the heuristic using a 2-deep software pipeline, 2 accelerators, and a BRAM size of 2048. Scaling is effec-

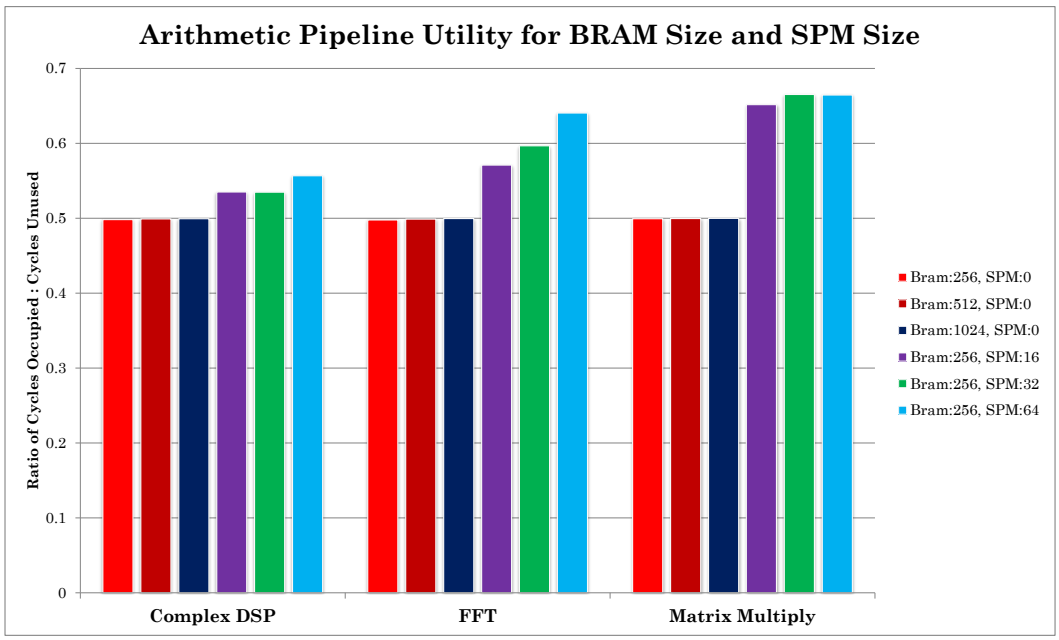
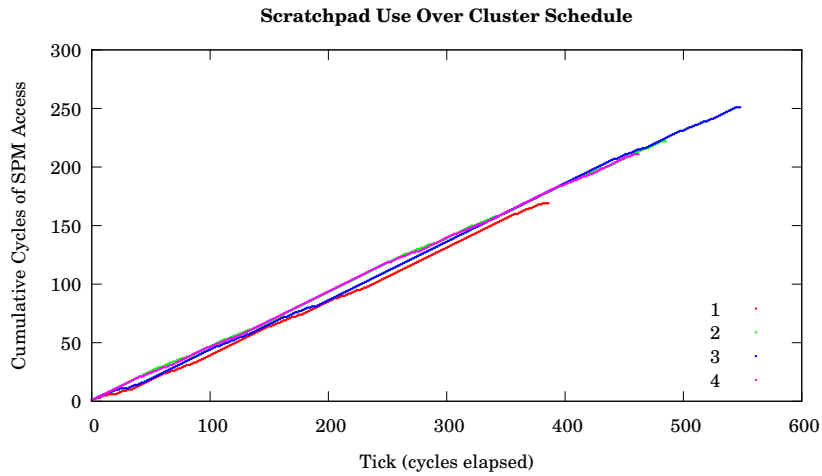
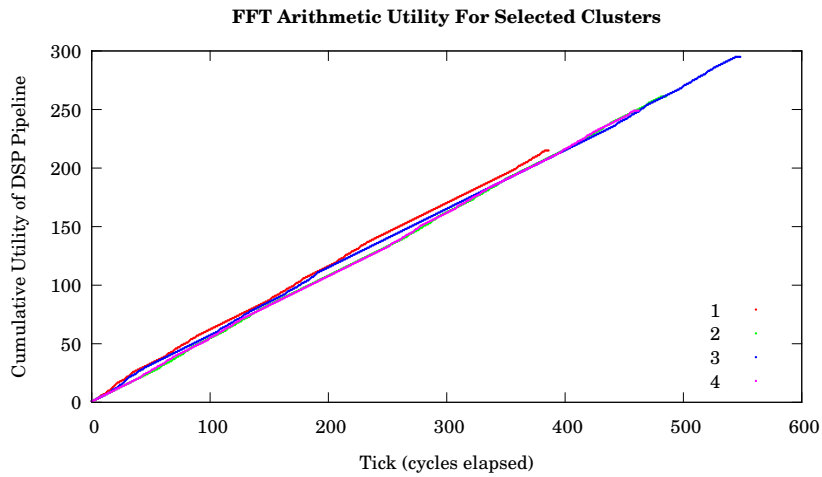


Figure 8.13: For each application, the utility represents the ratio of the used versus idle cycles in the arithmetic pipeline. As the BRAM size is varied, the utility remains relatively constant. Introducing a small scratchpad immediately increases utility; scratchpad size has a much greater impact on performance than BRAM size does.



(a) SPM utility over time



(b) Arithmetic utility over time

Figure 8.14: Visualization of scratchpad memory (SPM) and arithmetic pipeline utility for the same selected clusters of an FFT1024. The two are highly correlated: SPM size correlates strongly with higher performance, since fast access to operands allows the pipeline to remain busy.

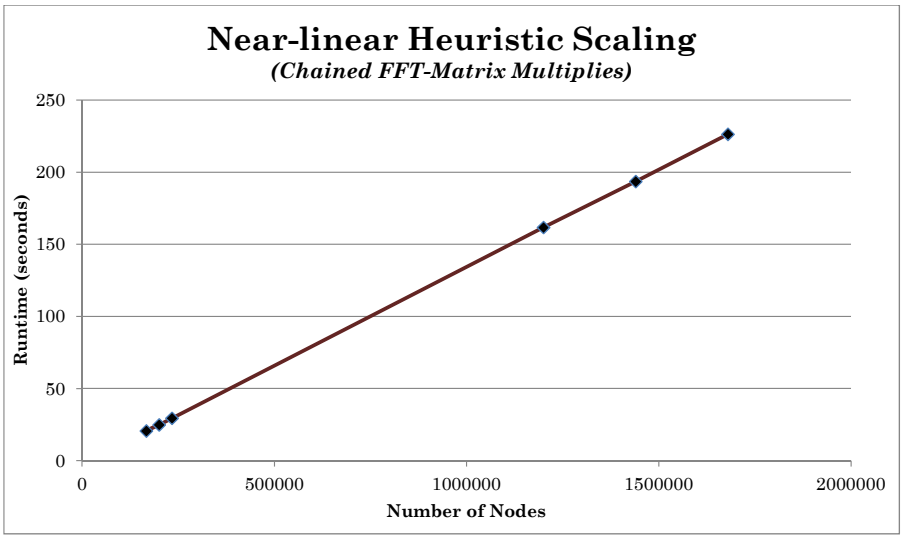


Figure 8.15: Heuristic runtime up to millions of nodes

tively linear, allowing this methodology to reach unprecedented design scales. A ~1.7 million node graph was clustered and scheduled with full memory constraints under 5 minutes.

8.6 Designer Insight

Automation exposes real system costs early in the design process through rapid design evaluation. The selected accelerator architecture suffered from memory port bottlenecks, illustrated by performance invariance to BRAM size. An alternative strategy using a scratchpad memory (even a relatively small one) is a significantly more effective use of available hardware resources. Instead of devoting more area to the BRAM, it can be routed towards much smaller scratchpad memory. Automation at these scales clearly grants designers insight into the problems they solve, and can handle capacity constraints and enormous problem sizes.

Chapter 9

Conclusions

Heterogeneous cosynthesis is a growing field, and this research has opened new potential to fluidly explore the architectural optimization space. Increased system heterogeneity brings with it unprecedented performance potential; the presented robust methodology provides a way to adapt to changing platforms without compromising performance. The hierarchical transactional framework lays groundwork for allowing rich specification of cosynthesis-targeted applications, leveraging a high degree of abstraction that keeps design expression concise. Its unique approach balances solutions for managing complex concurrent, latency tolerant tasks, with the ability to realize high performance implementations. This dissertation has demonstrated analysis of a control-dominated application, and full realization of a data-dominated high-performance signal processing suite.

Algorithms for mapping onto different semantic models underscore its ability to unify differing semantics in heterogeneous systems. A full suite of compiler tools enable practical design. At the same time, the semantics guide the designer towards specifications that do not hinder synthesis tools. Otherwise cumbersome

verification is made tenable through automated race condition identification, ameliorating common hurdles in ensuring concurrent tasks execute correctly.

Intrinsic flexibility brought about by state & control mobility creates opportunities to evaluate design configurations which were previously difficult to achieve automatically. By looking at the problem from an automation point of view, and structuring its semantics from the ground up, this research was able to identify and distill the core components required to allow truly automated cosynthesis.

9.1 Hierarchical Transaction Solutions

The hierarchical transactional semantic model is built on a set of core philosophies that bring state management to the forefront, encapsulating data with its associated control in a way that opens a vast array of opportunities in language design, verification, and synthesis. This novel, ground-up semantic model was exercised with a rich set of tools to specify, compile, analyze, simulate, and synthesize designs. It meets the four cosynthesis goals laid out at the start.

The Hierarchical Transactional Language (HTL) supports clear, concise application specification that communicates designer intent. A rich expression syntax keeps the code readable yet compact, removing cumbersome implementation of low level architectural details. Careful language design allows a compiler to leverage an intelligent type inference system for powerful compile-time parameterization. It efficiently captures semantics underpinning its data and execution models, reflecting the model's unique properties while retaining syntactic familiarity to ex-

isting languages. Specifications of both a processor and FFT are understandable, reinforcing the value of a novel language-based approach.

Practical applicability of a language depends on the usefulness of its associated tools. The compiler directly addresses correctness via behavioral simulation and fast analysis tools. Through conventional means, race conditions are difficult to identify, with a tapestry of solutions stemming from decades of research. The presented race condition detection mechanism rapidly isolates concurrency bugs in the face of latency tolerant behavior. By leveraging the unique encapsulation properties of hierarchical transactions, this otherwise complex and untenable problem is reduced to an elegant, pseudo-linear solution. Additionally, the issue of closure opens the opportunity for automated closure logic synthesis: tools can intelligently avoid synchronization mechanisms that may affect performance. They no longer need to synthesize against worst-case behavior. Through a single graph traversal, the compiler learns every possible closure condition.

Heterogeneous applications span the gamut from control- to data-dominated designs. Processor and signal processing algorithm implementations bookend this spectrum to exercise the benefits of the semantic model. A full ISA-compatible implementation of an Texas Instruments MSP430 proves that rich control semantics are both expressible and executable. The race condition tool quickly discovered an unexpected concurrent write ordering situation that was resolved in minutes. In a more qualitative sense, specification of the processor was straightforward – the flow of control is generally understandable, in large part due to code compactness and a single control semantic (tokens).

At the computational end of application styles, the language and model were capable of concisely specifying and synthesizing high-arithmetic applications commonly found in signal processing. Iterators lay the scaffolding for well-behaved data-flow graphs by codifying repetitive behavior. The compact, concise FFT specification proves that composing iterators sufficiently expresses complex indexing.

A data-flow graph-based solution automatically realizes efficient high-performance designs. Iterators are cast into well-defined data-flow graphs. They are then scheduled to reach unprecedented scales (scaling to millions of nodes). The methodology exposes system costs early in the design process, allowing focused improvements on resource constraints contributing most to performance bottlenecks.

9.2 Open Problems

Cosynthesis is an enormous problem space, and while hierarchical transactions have laid a solid foundation, there are outstanding opportunities to exercise the myriad aspects of this methodology. The largest area for research is in efficient control synthesis – arguably one of the most difficult elements to correct heterogeneous system design lies in orchestrating multiple components. Since the hierarchical transactional model codifies and encapsulates all state, there is potential to apply architectural synthesis in the form of automatic cache construction and distributed component synchronization.

Discussion of software scheduling has largely been omitted due to focus on optimization techniques and hardware targets. In the simplest implementation,

scheduling software transactions is similar to existing task scheduling solutions. When dealing with multiple software threads or processors, however, transactional encapsulation is invaluable in managing shared memory. The traditional locking/synchronization mechanisms (e.g. mutex and semaphore-based solutions) can be created only where necessary.

Bibliography

[1] K. Agrawal, J.T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174. ACM, 2008.

[2] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory models for open-nested transactions. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 70–81, New York, NY, USA, 2006. ACM.

[3] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software Architecture Optimization Methods: A Systematic Literature Review. *Software Engineering, IEEE Transactions on*, 39(5):658–683, May 2013.

[4] A. Allahverdi, C.T. Ng, T.C.E. Cheng, and M. Y. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985 – 1032, 2008.

[5] K. R. Apt, N. Francez, and W. P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):359–385, 1980.

[6] P. Arato, S. Juhasz, Z.A Mann, A Orban, and D. Papp. Hardware-software partitioning in embedded system design. In *Intelligent Signal Processing, 2003 IEEE International Symposium on*, pages 197–202, Sept 2003.

[7] K. Asanovic. Transactors for parallel hardware and software co-design. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 140 –142, nov. 2007.

[8] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies. In *Hardware/Software Codesign, 1997.(CODES/CASHE'97), Proceedings of the Fifth International Workshop on*, pages 161–165. IEEE, 1997.

[9] F. Balarin and R. Passerone. Specification, Synthesis, and Simulation of Transactor Processes. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(10):1749 –1762, oct. 2007.

[10] F. Balarin, Y. Watanabe, H. Hsieh, et al. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[11] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[12] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[13] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[14] J. Castrillon, W. Sheng, and R. Leupers. Trends in embedded software synthesis. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 347–354. IEEE, 2011.

[15] K. S. Chatha and R. Vemuri. MAGELLAN: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proceedings of the ninth international symposium on Hardware/software codesign*, CODES '01, pages 42–47, New York, NY, USA, 2001. ACM.

[16] S. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems–A Brief Survey. Technical report, Amherst, MA, USA, 1987.

[17] P.H. Chou, R.B. Ortega, and G. Borriello. The Chinook hardware/software co-synthesis system. In *Proceedings of the 8th international symposium on System synthesis*, pages 22–27. ACM, 1995.

[18] E.G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):78, 1971.

[19] J. Cong, B. Liu, S. Neuendorffer, et al. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[20] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[21] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of synchronous elastic circuits. In *TAU'06: Proceedings of the ACM/IEEE International Workshop on Timing Issues 2006*. Citeseer, 2006.

[22] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 657–662, New York, NY, USA, 2006. ACM.

[23] L.A. Cortés, P. Eles, and Z. Peng. A Survey on Hardware/Software Codesign Representation Models. Technical report, Department of Computer and Information Science Linköping University, S-581 83 Linköping, Sweden, June 1999.

[24] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on computers*, 36(5):547–553, 1987.

[25] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: hardware-software co-synthesis of embedded systems. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 703–708, New York, NY, USA, 1997. ACM.

[26] B.P. Dave and N.K. Jha. COHRA: hardware-software cosynthesis of hierarchical heterogeneous distributed embedded systems. *Computer-Aided Design*

of Integrated Circuits and Systems, IEEE Transactions on, 17(10):900 –919, oct 1998.

[27] G. De Micheli. Hardware synthesis from C/C++ models. In *Proceedings of the conference on Design, automation and test in Europe*, page 80. ACM, 1999.

[28] R.P. Dick and N.K. Jha. CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 62–67. ACM, 1998.

[29] S.A. Edwards. The challenges of hardware synthesis from C-like languages. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 66–67. IEEE, 2005.

[30] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. *Design Automation for Embedded Systems*, 2(1):5–32, 1997.

[31] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. SPECC: Specification Language and Methodology. 2000.

[32] W.M. Gentleman and G. Sande. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578. ACM, 1966.

[33] A. Gerstlauer, C. Haubelt, A.D. Pimentel, et al. Electronic System-Level Synthesis Methodologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1517 –1530, Oct. 2009.

[34] R.K. Gupta, Jr. Claudionor, N.C., and G. De Micheli. Program implementation schemes for hardware-software systems. *Computer*, 27(1):48–55, Jan 1994.

[35] R.K. Gupta and G. De Micheli. *Co-synthesis of hardware and software for digital embedded systems*, volume 4. Kluwer academic publishers Boston, MA, 1995.

[36] S. Ha, S. Kim, C. Lee, et al. PeaCE: A hardware-software codesign environment for multimedia embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):24, 2007.

[37] L. Hammond, V. Wong, M. Chen, et al. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, volume 32, page 102. IEEE Computer Society, 2004.

[38] C. Haubelt, J. Falk, J. Keinert, et al. A SystemC-based design methodology for digital signal processing systems. *EURASIP J. Embedded Syst.*, 2007(1):15–15, January 2007.

[39] C. Haubelt, J. Teich, K. Richter, and R. Ernst. System design for flexibility. In *Proceedings of the conference on Design, automation and test in Europe*, page 854. IEEE Computer Society, 2002.

[40] J. Henkel. A Low Power Hardware/Software Partitioning Approach for Core-based Embedded Systems. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 122–127, New York, NY, USA, 1999. ACM.

[41] J. Henkel and R. Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(2):273–289, April 2001.

[42] J. Henkel, R. Ernst, U. Holtmann, and T. Benner. Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 96–100. IEEE Computer Society Press, 1994.

[43] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[44] W. Herroelen, B. De Reyck, and E. Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4):279 – 302, 1998.

[45] J.I Hidalgo and J. Lanchares. Functional partitioning for hardware-software codesign using genetic algorithms. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, pages 631–638, Sept 1997.

[46] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[47] B. Holland, M. Vacas, V. Aggarwal, et al. Survey of C-based application mapping tools for reconfigurable computing. In *Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD'05)*, 2005.

[48] G. Hoover. *Distributed Control For Embedded System Design*. PhD thesis, University of California, Santa Barbara, Santa Barbara, California, 2008.

[49] G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 306–311, New York, NY, USA, 2008. ACM.

[50] T. Horder and K. Rothermel. Concurrency control issues in nested transactions. *The VLDB Journal–The International Journal on Very Large Data Bases*, 2(1):39–74, 1993.

[51] A. Jantsch, P. Ellervee, A. Hemani, J. Öberg, and H. Tenhunen. Hardware/Software Partitioning and Minimizing Memory Interface Traffic. In *Proceedings of the Conference on European Design Automation*, EURO-DAC '94, pages 226–231, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[52] A. Kalavade and E. A. Lee. A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem. In *Proceedings of the 3rd International Workshop on Hardware/Software Co-design*,

CODES '94, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[53] J. Keinert, M. Streubühorbar, T. Schlichter, et al. SystemCoDesigner–an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1:1–1:23, January 2009.

[54] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[55] V. Kianzad and S.S. Bhattacharyya. CHARMED: a multi-objective co-synthesis framework for multi-mode embedded systems. In *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pages 28–40, Sept 2004.

[56] D. Kim and S. Ha. Static analysis and automatic code synthesis of flexible FSM model. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 161–165. ACM, 2005.

[57] M. Kim, S. Banerjee, N. Dutt, and N. Venkatasubramanian. Energy-aware cosynthesis of real-time multimedia applications on MPSoCs using heterogeneous scheduling policies. *ACM Trans. Embed. Comput. Syst.*, 7:9:1–9:19, January 2008.

[58] M. King, N. Dave, et al. Automatic generation of hardware/software interfaces. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 325–336. ACM, 2012.

[59] S. Klaus, S. Huss, and T. Trautmann. Automatic Generation of Scheduled SystemC Models of Embedded Systems from Extended Task Graphs. In Eugenio Villar and Jean Mermet, editors, *System Specification and Design Languages*, pages 207–217. Springer US, 2003.

[60] P.V. Knudsen and J. Madsen. PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 85–, Washington, DC, USA, 1996. IEEE Computer Society.

[61] N. Krivokapić, A. Kemper, and E. Gudes. Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis. *The VLDB journal*, 8(2):79–100, 1999.

[62] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor. A transaction-based unified simulation/emulation architecture for functional verification. In *Design Automation Conference, 2001. Proceedings*, pages 623 – 628, 2001.

[63] E.A. Lee and II John. Overview of the ptolemy project, 1999.

[64] M. López-Vallejo and J. C. López. On the Hardware-software Partitioning Problem: System Modeling and Partitioning Techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):269–297, July 2003.

[65] M. Lukasiewycz, M. Streubuhr, M. Glass, C. Haubelt, and J. Teich. Combined system synthesis and communication architecture exploration for MP-SoCs. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 472–477, april 2009.

[66] R. Lysecky and F. Vahid. A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 18–23 Vol. 1, March 2005.

[67] B. Meals. Hierarchical Decomposition Algorithm for Hardware/Software Partitioning. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, pages 18–23, New York, NY, USA, 2006. ACM.

[68] R. Niemann and P. Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.

[69] H. Nikolov, M. Thompson, T. Stefanov, et al. Daedalus: Toward Composable Multimedia MP-SoC Design. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM.

[70] H. Oh and S. Ha. Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 133–138. ACM, 2002.

[71] C.S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339–353, 2009.

[72] S. Pasricha, N. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane. Floorplan-aware automated synthesis of bus-based communication architectures. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 565–570, New York, NY, USA, 2005. ACM.

[73] H.D. Patel, S.K. Shukla, E. Mednick, and R.S. Nikhil. A rule-based model of computation for SystemC: integrating SystemC and Bluespec for co-design. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 39 –48, july 2006.

[74] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[75] G. Quan, X.S. Hu, and G. Greenwood. Preference-driven hierarchical hardware/software partitioning. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 652–657, 1999.

[76] D. Saha, R.S. Mitra, and A Basu. Hardware software partitioning using genetic algorithm. In *VLSI Design, 1997. Proceedings., Tenth International Conference on*, pages 155–160, Jan 1997.

[77] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[78] L. Séméria and G. De Micheli. SpC: synthesis of pointers in C application of pointer analysis to the behavioral synthesis from C. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 340–346. IEEE, 1998.

[79] M. B. Srivastava and R. W Brodersen. Rapid-prototyping of hardware and software in a unified framework. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pages 152–155. IEEE, 1991.

[80] J. Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.

[81] Texas Instruments. *MSP430i2xx Family User's Guide*, August 2014.

[82] M. Thompson, H. Nikolov, T. Stefanov, et al. A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '07, pages 9–14, New York, NY, USA, 2007. ACM.

[83] F. Vahid, D. D. Gajski, and J. Gong. A Binary-constraint Search Algorithm for Minimizing Hardware During Hardware/Software Partitioning. In *Proceedings of the Conference on European Design Automation*, EURO-DAC '94, pages 214–219, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[84] T. Wiangtong, P.Y.K. Cheung, and W. Luk. Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware/Software Codesign. *Design Automation for Embedded Systems*, 6(4):425–449, 2002.

[85] F. Winterstein, S. Bayliss, and G.A Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 362–365, Dec 2013.

[86] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38 – 43, april 2003.

[87] W. Wolf. Design Challenges in Multiprocessor Systems-on-Chip. In Bernd Kleinjohann, Lisa Kleinjohann, Ricardo Machado, Carlos Pereira, and P. Thiagarajan, editors, *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, volume 225 of *IFIP International Federation for Information Processing*, pages 1–8. Springer Boston, 2006.

[88] W. Wolf, AA Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, Oct 2008.

[89] N.S. Woo, A.E. Dunlop, and W. Wolf. Codesign from cospecification. *Computer*, 27(1):42 –47, jan 1994.

[90] Xilinx. *Vivado Design Suite User Guide: Designing With IP*, October 2013.

[91] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 183–188, New York, NY, USA, 2002. ACM.

Appendix A

HTL Grammar

⟨*spec*⟩ ::= ⟨*stmt*⟩*

⟨*rule-def*⟩ ::= ‘`rule`’ ⟨*ident*⟩ ⟨*token-guard-list*⟩ ⟨*rule-input*⟩? ‘`:`’ suite

⟨*rule-input*⟩ ::= ⟨*rule-input-list*⟩

  |

⟨*rule-input-list*⟩ ::= ‘`<-`’ ‘`(`’ (⟨*ident*⟩ ‘`,`’)* ‘`)`’

  |

⟨*token-guard-list*⟩ ::= ⟨*empty*⟩

  | ‘`(`’ ⟨*token-guard*⟩? ‘`)`’

⟨*token-guard*⟩ ::= ⟨*token-guard-or*⟩

⟨*token-guard-or*⟩ ::= ⟨*token-guard-and*⟩ (‘`or`’ ⟨*token-guard-and*⟩)*

⟨*token-guard-and*⟩ ::= ⟨*token-atom*⟩ (‘`and`’ ⟨*token-atom*⟩)*

⟨*token-atom*⟩ ::= ⟨*ident*⟩

  | ‘`(`’ ⟨*token-guard-or*⟩ ‘`)`’

⟨*stmt*⟩ ::= ⟨*simple-stmt*⟩

| ⟨*compound-stmt*⟩

⟨*simple-stmt*⟩ ::= ⟨*small-stmt*⟩ (‘`;`’ ⟨*small-stmt*⟩)* ⟨*newline*⟩

⟨*small-stmt*⟩ ::= ⟨*vardecl-stmt*⟩

| ⟨*assign-stmt*⟩

| ⟨*print-stmt*⟩

| ⟨*create-stmt*⟩

| ⟨*pass-stmt*⟩

| ⟨*return-stmt*⟩

| ⟨*import-stmt*⟩

| ⟨*modifier-stmt*⟩

⟨*compound-stmt*⟩ ::= ⟨*if-stmt*⟩

| ⟨*for-stmt*⟩

| ⟨*rule-def*⟩

| ⟨*class-def*⟩

| ⟨*call-stmt*⟩

⟨*vardecl-stmt*⟩ ::= ‘`var`’ ⟨*ident*⟩ (‘`,`’ ⟨*ident*⟩)* ‘`:`’ ⟨*ident*⟩ (‘`(`’ ⟨*arglist*⟩? ‘`)`’)?

⟨*assign-stmt*⟩ ::= ⟨*lvalue-list*⟩ ‘`=`’ ⟨*test-list*⟩

⟨*aug-assign*⟩ ::= ‘`+=`’ | ‘`-=`’ | ‘`*=`’ | ‘`/=`’ | ‘`%=`’ | ‘`&=`’ | ‘`|=`’ | ‘`^=`’ | ‘`<<=`’ | ‘`>>=`’

⟨*print-stmt*⟩ ::= ‘`print`’ ⟨*test-list*⟩ ⟨*create-stmt*⟩ ::= ‘`create`’ ⟨*ident*⟩ ⟨*pass-stmt*⟩

::= ‘`pass`’

⟨*flow-stmt*⟩ ::= ⟨*break-stmt*⟩

| ⟨*continue-stmt*⟩

| ⟨*return-stmt*⟩

⟨*return-stmt*⟩ ::= '`return`' ⟨*test-list*⟩ ⟨*continue-stmt*⟩ ::= '`continue`' ⟨*break-stmt*⟩

::= '`break`'

⟨*import-stmt*⟩ ::= '`import`' ⟨*import-item*⟩ ('`,`' ⟨*import-item*⟩)* ⟨*import-item*⟩ ::=

⟨*ident*⟩ ('`as`' ⟨*ident*⟩)?

⟨*modifier-stmt*⟩ ::= '`@`' ⟨*ident*⟩ ('`(`' ⟨*arglist*⟩? '`)`')?

⟨*compoint-stmt*⟩ ::= ⟨*if-stmt*⟩

| ⟨*for-stmt*⟩

| ⟨*rule-def*⟩

| ⟨*class-def*⟩

| ⟨*call-stmt*⟩

⟨*if-stmt*⟩ ::= '`if`' ⟨*test*⟩ '`:`' ⟨*suite*⟩ ⟨*elif-clause*⟩* ('`else`' '`:`' ⟨*suite*⟩)?

⟨*elif-clause*⟩ ::= '`elif`' test '`:`' ⟨*suite*⟩

⟨*for-stmt*⟩ ::= '`for`' ⟨*for-iterators*⟩ '`:`' ⟨*suite*⟩ ⟨*for-finally*⟩?

⟨*for-finally*⟩ ::= '`finally`' '`:`' ⟨*suite*⟩

⟨*for-iterators*⟩ ::= ⟨*iterator-spec*⟩ ('`||`' ⟨*iterator-spec*⟩)*

⟨*iterator-spec*⟩ ::= ⟨*lvalue-list*⟩ '`in`' ⟨*test-list*⟩

⟨*call-stmt*⟩ ::= '`call`' (⟨*lvalue-list*⟩ '`=`')? ⟨*test-list*⟩ ⟨*call-optional-suite*⟩

⟨*call-optional-suite*⟩ ::= '`:`' ⟨*call-suite*⟩

| NEWLINE

⟨*creates-token-rename*⟩ ::= ⟨*ident*⟩ (‘`->`’ ⟨*ident*⟩)

⟨*creates-stmt*⟩ ::= ‘`creates`’ ⟨*creates-token-name*⟩+

⟨*call-param-small-stmt*⟩ ::= ⟨*creates-stmt*⟩

| ⟨*pass-stmt*⟩

⟨*call-param-stmt*⟩ ::= ⟨*call-param-small-stmt*⟩ (‘`;`’ ⟨*call-param-small-stmt*⟩)*

⟨*call-suite*⟩ ::= ⟨*call-param-stmt*⟩

| NEWLINE INDENT ⟨*call-param-stmt*⟩+ DEDENT

⟨*suite*⟩ ::= ⟨*simple-stmt*⟩

| NEWLINE INDENT ⟨*stmt*⟩+ DEDENT

⟨*test*⟩ ::= ⟨*or-test*⟩

⟨*or-test*⟩ ::= ⟨*and-test*⟩ (‘`or`’ ⟨*and-test*⟩)*

⟨*and-test*⟩ ::= ⟨*not-test*⟩ (‘`and`’ ⟨*not-test*⟩)*

⟨*not-test*⟩ ::= ‘`not`’ ⟨*not-test*⟩

| ⟨*comparison*⟩

⟨*comparison*⟩ ::= ⟨*expr*⟩ (⟨*comp-op*⟩ ⟨*expr*⟩)*

⟨*comp-op*⟩ ::= ‘`<`’

| ‘`>`’

| ‘`==`’

| ‘`>=`’

| ‘`<=`’

| ‘`=<`’

| ‘`!=`’

| ‘`~=`’

⟨*expr*⟩ ::= ⟨*tuple-expr*⟩

⟨*tuple-expr*⟩ ::= ⟨*expr-t*⟩

| ‘`(`’ ⟨*expr-t*⟩ (‘`,`’ ⟨*expr-t*⟩)+ ‘`)`’

⟨*expr-t*⟩ ::= ⟨*xor-expr*⟩ (‘`|`’ ⟨*xor-expr*⟩)*

⟨*xor-expr*⟩ ::= ⟨*and-expr*⟩ (‘`^`’ ⟨*and-expr*⟩)*

⟨*and-expr*⟩ ::= ⟨*shift-expr*⟩ (‘`&`’ ⟨*shift-expr*⟩)*

⟨*shift-expr*⟩ ::= ⟨*arith-expr*⟩ ((‘`<<`’ | ‘`>>`’) ⟨*arith-expr*⟩)*

⟨*arith-expr*⟩ ::= ⟨*term*⟩ ((‘`+`’ | ‘`-`’) ⟨*term*⟩)*

⟨*term*⟩ ::= ⟨*factor*⟩ ((‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`//`’) ⟨*factor*⟩)*

⟨*factor*⟩ ::= ‘`+`’ ⟨*factor*⟩

| ‘`-`’ ⟨*factor*⟩

| ‘`~`’ ⟨*factor*⟩

| ⟨*trailer*⟩

⟨*trailer*⟩ ::= ⟨*atom*⟩ (‘`[`’ ⟨*subscript*⟩ ‘`]`’ | ‘`.`’ ⟨*ident*⟩ | ‘`(`’ ⟨*arg-list*⟩ ‘`)`’)

⟨*atom*⟩ ::= ‘`(`’ ⟨*test*⟩? ‘`)`’

| ⟨*ident*⟩

| ⟨*int*⟩

| ⟨*float*⟩

| ⟨*string*⟩+

| ‘`true`’ | ‘`false`’

⟨*subscript*⟩ ::= ⟨*test*⟩ (‘`:`’ test)*

⟨*expr-list*⟩ ::= (⟨*expr*⟩ ‘`,`’)+

⟨*test-list*⟩ ::= (⟨*test*⟩ ‘`,`’)+

⟨*lvalue-list*⟩ ::= ⟨*lvalue-item*⟩ (‘`,`’ ⟨*lvalue-item*⟩)*

⟨*lvalue-item*⟩ ::= ⟨*trailer*⟩

⟨*class-def*⟩ ::= ‘`class`’ ⟨*ident*⟩ (‘`(`’ ⟨*arg-list*⟩ ‘`)`’)? ⟨*rule-input*⟩? ‘`:`’ ⟨*suite*⟩

⟨*arg-list*⟩ ::= (⟨*argument*⟩ ‘`,`’)*

⟨*argument*⟩ ::= ⟨*ident*⟩ ‘`=`’ ⟨*test*⟩

| ⟨*test*⟩

Appendix B

FFT and MSP430 Code Listing

B.1 FFT HTL Code Listing

```
1  import math
2
3  class FFT<-(size, type):
4     rule fft<-(self, Input, Coefficients):
5        var stages: Int(math.log2(self.size))
6        var i_stages: IIntSequence(stages)
7        var intermediates: Array(self.size, self.type) #
            intermediate results
8        @unroll
9        @pipeline(Input, intermediates)
10       for stage in i_stages:
11
12          print "stage = ", stage
13
14          # Calculate the "skip" size for this stage
15          var cluster_size: Int(math.pow(2, stages-stage))
16          var rr_size:  Int(math.pow(2, stages-stage-1),
               verbose='rr_size: ')
17
18          # Instantiate iterators
19          var in_clustered:   IClustered(Input, cluster_size
               )
20          var coef_clustered: IClustered(Coefficients,
               cluster_size)
21          var out_clustered:  IClustered(intermediates,
               cluster_size)
22
23          # Iterate over the sub cluster
24          for   in_cluster in in_clustered || \
25                c_cluster in coef_clustered || \
```

```
26                  out_cluster in out_clustered:
27
28              # Now iterate within the group
29              var i_in:   IGroupedRoundRobin(in_cluster,
                    rr_size, rr_size)
30              var i_coef: IGroupedRoundRobin(c_cluster,
                    rr_size, rr_size)
31              var i_out:  IGroupedRoundRobin(out_cluster,
                    rr_size, rr_size)
32              var tmp: Bits(16)
33              for out0, out1 in i_out || x, y in i_in || coef
                    , nul in i_coef:
34                  out0 = x + coef*y
35                  out1 = x - coef*y
36          finally:
37              create tDone
38
39  rule Top:
40      var fft_size: Int(2048) # Parameterizable
41      var f: FFT(fft_size, Bits(16))
42      rule testbench1:
43          var k: Array(fft_size, Bits(16), preload='fft_inputs.
                txt')
44          call f.fft(Input=k, Coefficients=k)
```

B.2 MSP430 Code Listing

```
1   # This is only the control core
2   # There is no actual memory or register file implementation
3   # since I removed array indices completely.
4
5   class Instruction:
6       var v: Bits(48)
7       rule single_opcode<-(self):
8           return self.v[9:7]
9       rule single_bw<-(self):
10          return self.v[6]
11      rule single_ad<-(self):
12          return self.v[5:4]
13      rule single_dsreg<-(self):
14          return self.v[3:0]
15      rule is_single<-(self):
16          return self.v[15:10] == 4
17
18      rule opRRC<-(self):
19          return self.single_opcode() == 0
20      rule opSWPB<-(self):
21          return self.single_opcode() == 1
```

```
22      rule opRRA<-(self):
23          return self.single_opcode() == 2
24      rule opSXT<-(self):
25          return self.single_opcode() == 3
26      rule opPUSH<-(self):
27          return self.single_opcode() == 4
28      rule opCALL<-(self):
29          return self.single_opcode() == 5
30      rule opRETI<-(self):
31          return self.single_opcode() == 6
32
33      rule double_opcode<-(self):
34          return self.v[15:12]
35      rule double_sreg<-(self):
36          return self.v[11:8]
37      rule double_ad<-(self):
38          return self.v[7]
39      rule double_bw<-(self):
40          return self.v[6]
41      rule double_as<-(self):
42          return self.v[5:4]
43      rule double_dreg<-(self):
44          return self.v[3:0]
45      rule is_double<-(self):
46          return self.v[14] or self.v[15]
47
48      rule opMOV<-(self):
49          return self.double_opcode() == 4
50      rule opADD<-(self):
51          return self.double_opcode() == 5
52      rule opADDC<-(self):
53          return self.double_opcode() == 6
54      rule opSUBC<-(self):
55          return self.double_opcode() == 7
56      rule opSUB<-(self):
57          return self.double_opcode() == 8
58      rule opCMP<-(self):
59          return self.double_opcode() == 9
60      rule opDADD<-(self):
61          return self.double_opcode() == 10
62      rule opBIT<-(self):
63          return self.double_opcode() == 11
64      rule opBIC<-(self):
65          return self.double_opcode() == 12
66      rule opBIS<-(self):
67          return self.double_opcode() == 13
68      rule opXOR<-(self):
69          return self.double_opcode() == 14
70      rule opAND<-(self):
71          return self.double_opcode() == 15
72
73      rule opJNE<-(self): # JNE/JNZ
74          return self.jump_condition() == 0
```

```
75      rule opJEQ<-(self): # JEQ/JZ
76          return self.jump_condition() == 1
77      rule opJNC<-(self): # JNC/JLO
78          return self.jump_condition() == 2
79      rule opJC<-(self): # JC/JHS
80          return self.jump_condition() == 3
81      rule opJN<-(self):
82          return self.jump_condition() == 4
83      rule opJGE<-(self):
84          return self.jump_condition() == 5
85      rule opJL<-(self):
86          return self.jump_condition() == 6
87      rule opJMP<-(self):
88          return self.jump_condition() == 7
89
90      rule jump_opcode<-(self):
91          return self.v[15:13]
92      rule jump_condition<-(self):
93          return self.v[12:10]
94      rule jump_offset<-(self):
95          return (self.v[9:0]) | \
96              (self.v[9] << 10) | \
97              (self.v[9] << 11) | \
98              (self.v[9] << 12) | \
99              (self.v[9] << 13) | \
100             (self.v[9] << 14) | \
101             (self.v[9] << 15)
102
103     rule is_jump<-(self):
104         return self.v[15:13] == 1
105
106     rule next_word<-(self):
107         return self.v[31:15]
108     rule third_word<-(self):
109         return self.v[47:32]
110
111 class Status:
112
113     var carry: Bits(1)
114     var zero: Bits(1)
115     var negative: Bits(1)
116     var gie: Bits(1)
117     var overflow: Bits(1)
118
119     rule FullRegister<-(self):
120         return self.carry | (self.zero << 1) | (self.negative
                << 2) | (self.gie << 3) | (self.overflow << 8)
121
122 class Memory:
123     var data: Addressable(size=65535, type=Bits(16))
124
125 rule TestbenchEnvironment:
126
```

```
127     var memory: Memory()
128     var instruction_limit: Bits(16)
129     var pc: Bits(16)
130
131     rule setupInstructionStream:
132        pc = 0
133        create tInitialize
134
135     var registers: Addressable(size=16, type=Bits(16))
136     var sp: Bits(16)
137     var sr: Bits(16)
138     var status: Status()
139     var pc_offset: Bits(16)
140
141     rule initialize(tInitialize):
142        sp = 0
143        sr = 0
144        status.carry = 0
145        status.zero = 0
146        status.negative = 0
147        status.gie = 0
148        status.overflow = 0
149        create tGetNextInstruction
150
151     rule getNextInstruction(tGetNextInstruction):
152        print "current pc=", pc
153        var i: Instruction()
154        if pc <= instruction_limit:
155           create tInstruction
156
157     rule MSP430(tInstruction):
158        var instr: Instruction()
159        rule GetFirstWord:
160           instr.v = memory.data[pc]
161           create tStartInstruction
162
163        rule InstructionStream(tStartInstruction):
164           var src_op, dest_op: Bits(16)
165           var src_pc_offset, dest_pc_offset: Bits(2)
166
167           var src, dest: Bits(4)
168           var smode, dmode: Bits(2)
169
170           rule Decode:
171              rule DetermineType:
172                 print "Instruction value: ", hex(instr.v)
173                 pc_offset = 1
174                 if instr.is_jump():
175                    print "  Jump operation"
176                    create tExecuteJump
177                 elif instr.is_single():
178                    print "  Single operation"
179                    create tDecodeSingle
```

```
180                elif instr.is_double():
181                   print "  Double operation"
182                   create tDecodeDouble
183                else:
184                   create tDecodeError
185
186             rule DecodeSingle(tDecodeSingle):
187                src = instr.single_dsreg()
188                smode = instr.single_ad()
189
190                if not instr.opRETI():
191                   create tGetSourceOperand
192                else:
193                   # For RETI instructions, we just execute
                         them directly. We just pretend
194                   # that the source operation was completed
                         by creating the token here.
195                   src_op = 0
196                   src_pc_offset = 1
197                   create tSourceReady
198
199             rule DecodeDouble(tDecodeDouble):
200                src = instr.double_sreg()
201                smode = instr.double_as()
202                create tGetSourceOperand
203
204                if instr.opMOV():
205                   create tDecodeMOV
206                else:
207                   dest = instr.double_dreg()
208                   dmode = instr.double_ad()
209                   create tGetDestOperand
210
211             rule DecodeMOV(tDecodeMOV):
212                # If it's a MOV instruction, we do not need
                      to read the destination
213                # so we just create the token here.
214                if smode == 1:
215                   dest_pc_offset = 1
216                else:
217                   dest_pc_offset = 0
218                dest_op = 0
219                create tDestReady
220
221          # end rule Decode
222
223          rule GetSourceOperand(tGetSourceOperand):
224             var srcaddr: Bits(16)
225
226             rule DetermineSourceAddressing:
227                # smode == 0: direct register smode
```

```
228                 # smode == 1: indirect, register + offset,
                        where the offset is in the next
                        instruction word
229                 #           unless the register is SR (
                        status reg): then it's an absolute
                        address
230                 # smode == 2: indirect register, but without
                         an offset
231                 # smode == 3: used with regular registers is
                         the same as smode==2, but it
                        autoincrements
232                 #           by 1 for byte instructions and
                        by 2 for word instructions
233                 #            When used with PC, the next
                        instruction word will be used as an
                        immediate constant
234                 src_pc_offset = 0
235
236                 # first deal with constant generation
237                 if (src == 2 and smode > 1) or src == 3:
238                     # note: if src==2 (status reg) and smode
                            (AS) == 01, then it's absolute smode
239                     #       if src==2 (SR) and smode (AS) ==
                            0, then it's a regular status reg read
240                     #       otherwise: it's the constant
                            generator
241                     create tConstantGenerator
242                 else:
243                     if smode == 0:
244                         print "     Source is register, r:",
                                src
245                         create tSourceIsRegister
246                     elif smode == 1:
247                         if src == 2: # status register
248                             print "     Source is absolute"
249                             create tSourceIsAbsolute
250                         else:
251                             print "     Source is indexed"
252                             create tSourceIsIndexed
253                     elif smode == 2:
254                         print "     Source is indirect"
255                         create tSourceIsIndirect
256                     elif smode == 3:
257                         if src == 0: # PC -> immediate
258                             print "     Source is immediate"
259                             create tSourceIsImmediate
260                         else:
261                             print "     Source is indirect
                                    autoincrement"
262                             create tSourceIsIndirect
263
264             rule SourceRegister(tSourceIsRegister):
265                 if src == 0: # PC
```

```
266                     src_op = pc
267                     create tSourceDone
268                 elif src == 1: # Stack Pointer
269                     src_op = sp
270                     create tSourceDone
271                 elif src == 2: # Status Pointer
272                     src_op = status.FullRegister()
273                     create tSourceDone
274                 else:
275                     # regular register lookup
276                     src_op = registers[src]
277                     create tSourceDone
278
279             rule SourceAbsolute(tSourceIsAbsolute):
280                 # address is stored in the next word
281                 srcaddr = memory.data[pc + 1]
282                 src_pc_offset = src_pc_offset + 1
283                 create tMemoryLookupSource
284
285             rule SourceIndexed(tSourceIsIndexed):
286                 var index_offset: Bits(16)
287                 var reg_offset: Bits(16)
288                 rule SourceIdxReadIndex:
289                     index_offset = memory.data[pc + 1]
290                     create tSourceIdxIndexReady
291                 rule SourceIdxReg:
292                     reg_offset = registers[src]
293                     create tSourceIdxRegReady
294                 rule SourceIdxMemRead(tSourceIdxIndexReady
                        and tSourceIdxRegReady):
295                     src_op = memory.data[reg_offset +
                            index_offset]
296                     src_pc_offset = src_pc_offset + 1
297                     create tSourceDone
298
299             rule SourceIndirect(tSourceIsIndirect):
300                 var addr: Bits(16)
301                 rule SourceIndReadAddr:
302                     addr = memory.data[pc + 1]
303                     create tSourceIndMemRead
304                 rule SourceIndMemRead(tSourceIndMemRead):
305                     src_op = memory.data[addr]
306                     src_pc_offset = src_pc_offset + 1
307                     create tSourceDone
308
309             rule SourceImmediate(tSourceIsImmediate):
310                 src_op = memory.data[pc + 1]
311                 src_pc_offset = src_pc_offset + 1
312                 create tSourceDone
313
314             rule ConstantGenerator(tConstantGenerator):
315                 if src == 2:
316                     # status register
```

```
317                     if smode == 0:
318                         src_op = status.FullRegister()
319                         print "Error: This should not have
                                been reached."
320                         create tDecodeError
321                     # actual constants (result is determined
                            by addressing smode)
322                     elif smode == 1:
323                         src_op = 0
324                     elif smode == 2:
325                         src_op = 4
326                     elif smode == 3:
327                         src_op = 8
328                 elif src == 3:
329                     # more constants, again determined by
                            addressing smode
330                     if smode == 0:
331                         src_op = 0
332                     elif smode == 1:
333                         src_op = 1
334                     elif smode == 2:
335                         src_op = 2
336                     elif smode == 3:
337                         src_op = 0xFFFF
338                 create tSourceDone
339
340             rule SourceMemory(tMemoryLookupSource):
341                 src_op = memory.data[srcaddr]
342                 create tSourceDone
343
344             rule printSourceOperand(tSourceDone):
345                 print "   Source operand: ", src_op
346                 create tSourceReady
347
348         # end rule GetSourceOperand
349         rule GetDestOperand(tGetDestOperand):
350             var destaddr: Bits(16)
351
352             rule DetermineDestAddressing:
353                 dest_pc_offset = 0
354
355                 if dmode == 0:
356                     print "     Dest is register, r:", dest
357                     create tDestIsRegister
358                 elif dmode == 1:
359                     if dest == 2: # status register
360                         print "     Dest is absolute"
361                         create tDestIsAbsolute
362                     else:
363                         print "     Dest is indexed"
364                         create tDestIsIndexed
365                 else:
366                     print "Dest is UNKNOWN"
```

```
367                     create tDecodeError
368
369                 rule DestRegister(tDestIsRegister):
370                     if dest == 0: # PC
371                         dest_op = pc
372                         create tDestReadDone
373                     elif dest == 1: # Stack Pointer
374                         dest_op = sp
375                         create tDestReadDone
376                     elif dest == 2: # Status Pointer
377                         dest_op = status.FullRegister()
378                         create tDestReadDone
379                     else:
380                         # regular register lookup
381                         dest_op = registers[dest]
382                         create tDestReadDone
383
384                 rule DestAbsolute(tDestIsAbsolute):
385                     # address is stored in the next word
386                     destaddr = memory.data[pc + 1]
387                     dest_pc_offset = 1
388                     create tMemoryLookupDest
389
390                 rule DestIndexed(tDestIsIndexed):
391                     var index_offset: Bits(16)
392                     var reg_offset: Bits(16)
393                     rule DestIdxReadIndex:
394                         index_offset = memory.data[pc + 1]
395                         create tDestIdxIndexReady
396                     rule DestIdxReg:
397                         reg_offset = registers[dest]
398                         create tDestIdxRegReady
399                     rule DestIdxMemRead(tDestIdxIndexReady and
                            tDestIdxRegReady):
400                         dest_op = memory.data[reg_offset +
                                index_offset]
401                         dest_pc_offset = 1
402                         create tDestReadDone
403
404                 rule DestMemory(tMemoryLookupDest):
405                     dest_op = memory.data[destaddr]
406                     create tDestReadDone
407
408                 rule printDestOperand(tDestReadDone):
409                     print "Dest operand: ", dest_op
410                     create tDestReady
411             # end GetDestOperand
412
413             rule ExecuteJump(tExecuteJump):
414                 # ====================
415                 #    Jump Operations
416                 # ====================
417                 print "JMP Operation"
```

```
418             if instr.opJNE() and status.zero == 0:
419                 pc_offset = instr.jump_offset() + 1
420             elif instr.opJEQ() and status.zero == 1:
421                 pc_offset = instr.jump_offset() + 1
422             elif instr.opJNC() and status.carry == 0:
423                 pc_offset = instr.jump_offset() + 1
424             elif instr.opJC() and status.carry == 1:
425                 pc_offset = instr.jump_offset() + 1
426             elif instr.opJN() and status.negative == 1:
427                 pc_offset = instr.jump_offset() + 1
428             elif instr.opJGE() and status.negative ==
                    status.overflow:
429                 pc_offset = instr.jump_offset() + 1
430             elif instr.opJL() and status.negative != status
                    .overflow:
431                 pc_offset = instr.jump_offset() + 1
432             elif instr.opJMP():
433                 pc_offset = instr.jump_offset() + 1
434             else:
435                 pc_offset = 1
436             create tExecutionDone
437 
438         var write_src     : Bits(4)
439         var write_mode    : Bits(2)
440         var write_bw       : Bits(1)
441         var write_pc_offset: Bits(2)
442         var result        : Bits(17)
443 
444         rule Execute(tSourceReady and tDestReady):
445 
446             rule DetermineExecutionType:
447                 # first combine pc offsets
448                 pc_offset = pc_offset + dest_pc_offset +
                        src_pc_offset
449                 print "pc_offset =", pc_offset, "
                        dest_pc_offset =", dest_pc_offset, "
                        src_pc_offset =", src_pc_offset
450                 if instr.is_single():
451                     create tExecuteSingle
452                 elif instr.is_double():
453                     create tExecuteDouble
454                 else:
455                     create tDecodeError
456 
457             # ====================
458             #    Single Operations
459             # ====================
460             rule ExecuteSingle(tExecuteSingle):
461                 rule DetermineSingleOperation:
462                     if instr.opRRC():
463                         create tRRC
464                     elif instr.opSWPB():
465                         create tSWPB
```

```
466                     elif instr.opRRA():
467                         create tRRA
468                     elif instr.opSXT():
469                         create tSXT
470                     elif instr.opPUSH():
471                         create tPUSH
472                     elif instr.opCALL():
473                         create tCALL
474                     elif instr.opRETI():
475                         create tRETI
476                     else:
477                         create tDecodeError
478
479                 # Single ops
480                 rule SXT(tSXT):
481                     rule SignExtend:
482                         result = (src_op[7] << 15) | \
483                                  (src_op[7] << 14) | \
484                                  (src_op[7] << 13) | \
485                                  (src_op[7] << 12) | \
486                                  (src_op[7] << 11) | \
487                                  (src_op[7] << 10) | \
488                                  (src_op[7] << 9) | \
489                                  (src_op[7] << 8) | \
490                                  src_op[7:0]
491                         create tSXTStatus
492                     rule SXTStatus(tSXTStatus):
493                         status.carry = (result != 0)
494                         status.zero = (result == 0)
495                         status.negative = result[15]
496                         status.overflow = 0
497                         create tDestinationIsSource
498
499                 rule RRC(tRRC):
500                     rule ShiftAndRotate:
501                         if instr.single_bw(): # byte
502                             result = (status.carry << 7) | (
                                    src_op >> 1) & 0x7F
503                         else: # word
504                             result = (status.carry << 15) | (
                                    src_op >> 1)
505                         create tRRCStatus
506                     rule RRCStatus(tRRCStatus):
507                         status.carry = src_op[0] # carry
508                         status.zero = (result == 0)
509                         status.negative = result[15]
510                         status.overflow = 0
511                         create tDestinationIsSource
512
513                 rule RRA(tRRA):
514                     rule RotateRightArith:
515                         if instr.single_bw():
```

```
516                         result = (src_op & 0x80) | (src_op
                               [7:0] >> 1)
517                      else:
518                         result = (src_op & 0x8000) | (
                               src_op[15:0] >> 1)
519                      create tRRAStatus
520                   rule RRAStatus(tRRAStatus):
521                      status.carry = src_op[0]
522                      status.zero = (result == 0)
523                      if instr.single_bw():
524                         status.negative = result[7]
525                      else:
526                         status.negative = result[15]
527                      status.overflow = 0
528                      create tDestinationIsSource
529
530                rule SWPB(tSWPB):
531                   result = (src_op[7:0] << 8) | (src_op
                         [15:8] >> 8)
532                   create tDestinationIsSource
533
534                rule PUSH(tPUSH):
535                   sp = sp - 2
536                   memory.data[sp] = src_op
537                   create tExecutionDone
538
539                rule CALL(tCALL):
540                   sp = sp - 2
541                   memory.data[sp] = pc
542                   pc = src_op
543                   create tExecutionDone
544
545                rule RETI(tRETI):
546                   # restore status register
547                   var status_reg: Bits(16)
548                   rule RETILoad:
549                      status_reg = memory.data[sp]
550                      create tRETILoad
551                   rule RETIExc(tRETILoad):
552                      status.carry = status_reg[0]
553                      status.zero = status_reg[1]
554                      status.negative = status_reg[2]
555                      status.overflow = status_reg[8]
556
557                      pc = memory.data[sp + 2]
558                      sp = sp + 4
559                      create tExecutionDone
560
561             # ====================
562             #    Double Operations
563             # ====================
564             rule ExecuteDouble(tExecuteDouble):
565                rule DetermineDoubleOperation:
```

```
566                     if instr.opMOV():
567                         create tMOV
568                     elif instr.opADD():
569                         create tADD
570                     elif instr.opADDC():
571                         create tADDC
572                     elif instr.opSUB():
573                         create tSUB
574                     elif instr.opSUBC():
575                         create tSUBC
576                     elif instr.opDADD():
577                         create tDADD
578                     elif instr.opBIT():
579                         create tBIT
580                     elif instr.opCMP():
581                         create tCMP
582                     else:
583                         create tDecodeError
584
585                 # Double ops
586                 rule MOV(tMOV):
587                     print "MOV operation."
588                     result = src_op
589                     create tWriteDestination
590
591                 rule ADD(tADD):
592                     print "ADD operation. ", src_op, " + ",
                            dest_op
593                     result = src_op + dest_op
594                     create tUpdateStatusWriteDest
595
596                 rule ADDC(tADDC):
597                     print "ADDC operation."
598                     result = src_op + dest_op + status.carry
599                     create tUpdateStatusWriteDest
600
601                 rule SUB(tSUB):
602                     print "SUB operation."
603                     result = src_op - dest_op
604                     create tUpdateStatusWriteDest
605
606                 rule CMP(tCMP):
607                     print "CMP operation => ", src_op, "-",
                            dest_op, " = ", (src_op - dest_op)
608                     result = src_op - dest_op
609                     create tUpdateStatusWriteDest
610
611                 rule SUBC(tSUBC):
612                     print "SUBC operation."
613                     result = src_op - dest_op + status.carry
614                     create tUpdateStatusWriteDest
615
616                 rule DADD(tDADD):
```

```
617                     print "DADD operation."
618                     # Treat the number as a signed BCD
619                     var c0, c1, c2, c3: Bits(1)
620                     rule DADD_digit1:
621                        var m: Bits(4)
622                        rule DADD_d1_offset:
623                           m = src_op[3:0] + dest_op[3:0]
624                           create tDADD_d1_offset
625                        rule DADD_d1(tDADD_d1_offset):
626                           print "DEBUG m = ",m, "m <= 9:", m
                                 <= 9
627                           if m <= 9:
628                              result[3:0] = m
629                              c0 = 0
630                           else:
631                              result[3:0] = m - 10
632                              c0 = 1
633                           create tDADD_digit2
634                     rule DADD_digit2(tDADD_digit2):
635                        var m: Bits(4)
636                        rule DADD_d2_offset:
637                           m = src_op[7:4] + dest_op[7:4] + c0
638                           create tDADD_d2_offset
639                        rule DADD_d2(tDADD_d2_offset):
640                           if m <= 9:
641                              result[7:4] = m
642                              c1 = 0
643                           else:
644                              result[7:4] = m - 10
645                              c1 = 1
646                           if instr.double_bw(): # if it's a
                                 word...
647                              create tDADD_digit3
648                           else:
649                              create tWriteDestination
650                     rule DADD_digit3(tDADD_digit3):
651                        var m: Bits(4)
652                        rule DADD_d3_offset:
653                           m = src_op[11:8] + dest_op[11:8] +
                                 c1
654                           create tDADD_d3_offset
655                        rule DADD_d3(tDADD_d3_offset):
656                           if m <= 9:
657                              result[11:8] = m
658                              c2 = 0
659                           else:
660                              result[11:8] = m - 10
661                              c2 = 1
662                           create tDADD_digit4
663                     rule DADD_digit4(tDADD_digit4):
664                        var m: Bits(4)
665                        rule DADD_d4_offset:
```

```
666                         m = src_op[15:12] + dest_op[15:12]
                               + c2
667                         create tDADD_d4_offset
668                      rule DADD_d4(tDADD_d4_offset):
669                         if m <= 9:
670                            result[15:12] = m
671                         else:
672                            result[15:12] = m - 10
673                         create tUpdateStatusWriteDest
674
675                   rule BIT(tBIT):
676                      print "SUBC operation."
677                      result = src_op - dest_op + status.carry
678                      status.carry = result[16]
679                      create tUpdateStatusWriteDest
680
681                   rule UpdateStatusAndWriteDestination(
                         tUpdateStatusWriteDest):
682                      if instr.double_bw():
683                         status.zero = (result[15:0] == 0)
684                         status.carry = result[16]
685                         status.overflow = ((src_op[15] &
                            dest_op[15]) & ~result[15]) | ((~
                            src_op[15] & ~dest_op[15]) & result
                            [15])
686                         status.negative = result[15]
687                      else:
688                         status.zero = (result[7:0] == 0)
689                         status.carry = result[8]
690                         status.overflow = ((src_op[7] &
                            dest_op[7]) & ~result[7]) | ((~
                            src_op[7] & ~dest_op[7]) & result
                            [7])
691                         status.negative = result[7]
692                         result = result & 0xFF
693                      if not instr.opCMP():
694                         print "*** result = ", result
695                         create tWriteDestination
696                      else:
697                         create tExecutionDone
698
699                # ====================
700                #  Write Destination
701                # ====================
702                rule DestinationIsSource(tDestinationIsSource):
703                   write_src = instr.single_dsreg()
704                   write_mode = instr.single_ad()
705                   write_bw = instr.single_bw()
706                   write_pc_offset = 0
707                   create tWriteback
708
709                rule WriteDoubleResult(tWriteDestination):
710                   write_src = instr.double_dreg()
```

```
711                 write_mode = instr.double_ad()
712                 write_bw = instr.double_bw()
713                 write_pc_offset = 0
714                 create tWriteback
715
716             # end Execute
717
718             # write back
719             rule Writeback(tWriteback):
720                 rule DetermineWritebackMode:
721                     if (write_src == 2 and write_mode > 1) or
                            write_src == 3:
722                         create tCGError # attempt to write to the
                                CG
723                     else:
724                         if write_mode == 0:
725                             print "Writeback is register, r:",
                                    write_src
726                             create tWritebackIsRegister
727                         elif write_mode == 1:
728                             if write_src == 2: # status register
729                                 print "Writeback is absolute"
730                                 create tWritebackIsAbsolute
731                             elif write_src == 0: # pc
732                                 print "Writeback is symbolic"
733                                 create tWritebackIsSymbolic
734                             else:
735                                 print "Writeback is indexed"
736                                 create tWritebackIsIndexed
737
738                 rule WritebackIsRegister(tWritebackIsRegister):
739                     if write_src == 0: # PC
740                         pc = result
741                         create tWritebackDone
742                     elif write_src == 1: # Stack Pointer
743                         sp = result
744                         create tWritebackDone
745                     elif write_src == 2: # Status Pointer
746                         status.carry = result[0]
747                         status.zero = result[1]
748                         status.negative = result[2]
749                         status.gie = result[3]
750                         status.overflow = result[8]
751                         create tWritebackDone
752                     else:
753                         # regular register write
754                         registers[write_src] = result
755                         print "Writing ", result, " to registers
                                [", write_src, "]"
756                         create tWritebackDone
757
758                 rule WritebackIsAbsolute(tWritebackIsAbsolute):
759                     var addr: Bits(16)
```

```
760                 rule WritebackAbsReadAddr:
761                     addr = memory.data[pc + write_pc_offset]
762                     create tWritebackAbsReadMem
763                 rule WritebackAbsReadMem(
                        tWritebackAbsReadMem):
764                     memory.data[addr] = result
765                     print "Memory write", result, "to address
                            ", hex(addr)
766                     create tWritebackDone
767
768             rule WritebackIsSymbolic(tWritebackIsSymbolic):
769                 var index_offset: Bits(16)
770                 var addr: Bits(16)
771                 rule WritebackSymReadIndex:
772                     index_offset = memory.data[pc +
                            write_pc_offset]
773                     create tWritebackSymReadAddr
774                 rule WritebackSymReadAddr(
                        tWritebackSymReadAddr):
775                     addr = memory.data[pc + index_offset]
776                     create tWritebackSymReadMem
777                 rule WritebackSymReadMem(
                        tWritebackSymReadMem):
778                     memory.data[addr] = result
779                     print "Memory write", result, "to
                            symbolic address", hex(addr)
780                     create tWritebackDone
781
782             rule WritebackIsIndexed(tWritebackIsIndexed):
783                 var index_offset: Bits(16)
784                 var addr: Bits(16)
785                 rule WritebackIdxReadIndex:
786                     index_offset = memory.data[pc +
                            write_pc_offset]
787                     create tWritebackIdxReadAddr
788                 rule WritebackIdxReadAddr(
                        tWritebackIdxReadAddr):
789                     addr = registers[write_src] +
                            index_offset
790                     create tWritebackIdxReadMem
791                 rule WritebackIdxReadMem(
                        tWritebackIdxReadMem):
792                     memory.data[addr] = result
793                     print "Indexed memory.data write", result
                            , "to address", hex(addr)
794                     create tWritebackDone
795
796             rule WritebackCleanup(tWritebackDone):
797                 create tExecutionDone
798
799         # end rule InstructionStream
800
801         rule DecodeError(tDecodeError):
```

```
802             print "Decode error for instruction: ", instr.v,
                    hex(instr.v)
803
804         rule CGError(tCGError):
805             print "Decode error [write to CG] for instruction:
                     ", instr.v, hex(instr.v)
806
807         rule Cleanup(tExecutionDone):
808             create tInstructionDone
809
810         rule InstructionDone(tInstructionDone):
811             pc = pc + pc_offset
812             print "Instruction done. pc=", pc, "pc-offset=",
                    hex(pc_offset)
```

Appendix C

Tables of Matrix Multiplication Clustering

C.1 Matrix Multiplication Size Sweep Data

Software Pipeline Depth	# Accelerators	BRAM Size	SPM Size	N	M	P	# Nodes	Schedule Length	Clusters	Runtime (s)
2	2	128	8	8	8	12	1696	2321	2	0.28
2	2	128	16	8	8	12	1696	2321	2	0.27
2	2	128	32	8	8	12	1696	2321	2	0.28
2	2	128	64	8	8	12	1696	2321	2	0.27
2	2	256	8	8	8	12	1696	2317	1	0.27
2	2	256	16	8	8	12	1696	2317	1	0.26
2	2	256	32	8	8	12	1696	2317	1	0.26
2	2	256	64	8	8	12	1696	2317	1	0.26
2	2	512	8	8	8	12	1696	2317	1	0.26
2	2	512	16	8	8	12	1696	2317	1	0.27
2	2	512	32	8	8	12	1696	2317	1	0.27
2	2	512	64	8	8	12	1696	2317	1	0.26
2	2	128	8	10	2	16	692	972	2	0.11
2	2	128	16	10	2	16	692	972	2	0.10
2	2	128	32	10	2	16	692	972	2	0.11

Software Pipeline Depth	**# Accelerators**	**BRAM Size**	**SPM Size**	**N**	**M**	**P**	**# Nodes**	**Schedule Length**	**Clusters**	**Runtime (s)**
2	2	128	64	10	2	16	692	972	2	0.11
2	2	256	8	10	2	16	692	968	1	0.10
2	2	256	16	10	2	16	692	968	1	0.10
2	2	256	32	10	2	16	692	968	1	0.10
2	2	256	64	10	2	16	692	968	1	0.11
2	2	512	8	10	2	16	692	968	1	0.11
2	2	512	16	10	2	16	692	968	1	0.11
2	2	512	32	10	2	16	692	968	1	0.10
2	2	512	64	10	2	16	692	968	1	0.10
2	2	128	8	16	8	16	4352	6170	4	0.83
2	2	128	16	16	8	16	4352	6169	4	0.76
2	2	128	32	16	8	16	4352	6170	4	0.77
2	2	128	64	16	8	16	4352	6169	4	0.76
2	2	256	8	16	8	16	4352	6163	2	0.75
2	2	256	16	16	8	16	4352	6161	2	0.75
2	2	256	32	16	8	16	4352	6161	2	0.76
2	2	256	64	16	8	16	4352	6161	2	0.76
2	2	512	8	16	8	16	4352	6157	1	0.70
2	2	512	16	16	8	16	4352	6157	1	0.70
2	2	512	32	16	8	16	4352	6157	1	0.70
2	2	512	64	16	8	16	4352	6157	1	0.71
2	2	128	8	32	16	16	17152	25642	7	3.21
2	2	128	16	32	16	16	17152	25124	7	3.19
2	2	128	32	32	16	16	17152	24620	8	3.24
2	2	128	64	32	16	16	17152	24619	8	3.27
2	2	256	8	32	16	16	17152	25628	4	3.15
2	2	256	16	32	16	16	17152	25114	4	3.16
2	2	256	32	32	16	16	17152	24603	4	3.22
2	2	256	64	32	16	16	17152	24603	4	3.22
2	2	512	8	32	16	16	17152	25618	2	3.15
2	2	512	16	32	16	16	17152	25106	2	3.16
2	2	512	32	32	16	16	17152	24595	2	3.18
2	2	512	64	32	16	16	17152	24595	2	3.13

C.2 Software-Pipelining Data

Software Pipeline Depth	# Accel	BRAM Size	SPM Size	N	M	P	# Nodes	Schedule Length	Clusters	Runtime (s)	Average Utility
3	3	128	32	14	16	12	5792	8092	4	1.09	0.66
4	3	128	32	14	16	12	5792	16181	4	1.15	0.66
5	3	128	32	14	16	12	5792	16181	4	1.15	0.66
6	3	128	32	14	16	12	5792	16181	4	1.15	0.66
7	3	128	32	14	16	12	5792	24271	4	1.20	0.66
8	3	128	32	14	16	12	5792	24271	4	1.20	0.66
3	4	128	32	14	16	12	5792	8092	4	1.10	0.66
4	4	128	32	14	16	12	5792	8091	4	1.10	0.66
5	4	128	32	14	16	12	5792	16181	4	1.17	0.66
6	4	128	32	14	16	12	5792	16181	4	1.17	0.66
7	4	128	32	14	16	12	5792	16181	4	1.17	0.66
8	4	128	32	14	16	12	5792	16181	4	1.18	0.66
3	5	128	32	14	16	12	5792	8091	4	1.11	0.66
4	5	128	32	14	16	12	5792	8091	4	1.11	0.66
5	5	128	32	14	16	12	5792	8091	4	1.12	0.66
6	5	128	32	14	16	12	5792	16181	4	1.19	0.66
7	5	128	32	14	16	12	5792	16181	4	1.18	0.66
8	5	128	32	14	16	12	5792	16181	4	1.19	0.66
3	6	128	32	14	16	12	5792	8095	5	1.13	0.66
4	6	128	32	14	16	12	5792	8092	4	1.13	0.66
5	6	128	32	14	16	12	5792	8091	4	1.12	0.66
6	6	128	32	14	16	12	5792	8091	4	1.13	0.66
7	6	128	32	14	16	12	5792	16181	4	1.21	0.66
8	6	128	32	14	16	12	5792	16181	4	1.21	0.66
3	7	128	32	14	16	12	5792	8091	4	1.14	0.66
4	7	128	32	14	16	12	5792	8091	4	1.13	0.66
5	7	128	32	14	16	12	5792	8091	4	1.13	0.66
6	7	128	32	14	16	12	5792	8092	4	1.15	0.66
7	7	128	32	14	16	12	5792	8091	4	1.15	0.66
8	7	128	32	14	16	12	5792	16181	4	1.23	0.66
3	8	128	32	14	16	12	5792	8091	4	1.13	0.66

Software Pipeline Depth	**# Accel**	**BRAM Size**	**SPM Size**	**N**	**M**	**P**	**# Nodes**	**Schedule Length**	**Clusters**	**Runtime (s)**	**Average Utility**
4	8	128	32	14	16	12	5792	8091	4	1.15	0.66
5	8	128	32	14	16	12	5792	8091	4	1.14	0.66
6	8	128	32	14	16	12	5792	8091	4	1.16	0.66
7	8	128	32	14	16	12	5792	8091	4	1.14	0.66
8	8	128	32	14	16	12	5792	8092	4	1.15	0.66
3	3	128	32	14	16	12	5792	8092	4	1.09	0.66
4	3	128	32	14	16	12	5792	16181	4	1.15	0.66
5	3	128	32	14	16	12	5792	16181	4	1.15	0.66
6	3	128	32	14	16	12	5792	16181	4	1.15	0.66
7	3	128	32	14	16	12	5792	24271	4	1.20	0.66
8	3	128	32	14	16	12	5792	24271	4	1.20	0.66
3	4	128	32	14	16	12	5792	8092	4	1.10	0.66
4	4	128	32	14	16	12	5792	8091	4	1.10	0.66
5	4	128	32	14	16	12	5792	16181	4	1.17	0.66
6	4	128	32	14	16	12	5792	16181	4	1.17	0.66
7	4	128	32	14	16	12	5792	16181	4	1.17	0.66
8	4	128	32	14	16	12	5792	16181	4	1.18	0.66

Appendix D

Tables of FFT Clustering

D.1 FFT Size Sweep Data

Software Pipeline Depth	# Accelerators	BRAM Size	SPM Size	FFT Size	# Nodes	Schedule Length	Clusters	Runtime (s)
1	2	256	0	8192	114688	213195	49	23.87
1	2	256	8	8192	114688	205436	49	24.82
1	2	256	16	8192	114688	196926	49	24.13
1	2	256	32	8192	114688	191435	49	24.00
1	2	256	64	8192	114688	183804	49	23.59
1	2	512	0	8192	114688	213099	25	23.58
1	2	512	8	8192	114688	205107	25	24.29
1	2	512	16	8192	114688	196115	25	24.35
1	2	512	32	8192	114688	191371	25	24.21
1	2	512	64	8192	114688	183594	25	24.15
1	2	1024	0	8192	114688	213049	13	24.24
1	2	1024	8	8192	114688	204494	13	24.05
1	2	1024	16	8192	114688	196324	13	23.92
1	2	1024	32	8192	114688	191263	13	23.71
1	2	1024	64	8192	114688	183775	13	24.34

Appendix E

Tables of Large Pipelined DSP Example

The following tables list near-linear scalability up to million-node sizes for a multi-level pipelined complex DSP.

Software Pipeline Depth	# Accelerators	BRAM Size	SPM Size	P	Depth	# Nodes	Schedule Length	Clusters	Runtime (s)
2	2	2048	32	16	10	110080	166744	2	20.68
2	2	2048	32	16	12	132096	200092	2	24.92
2	2	2048	32	16	14	154112	233438	2	29.47
2	2	2048	32	32	10	788480	1199721	6	161.68
2	2	2048	32	32	12	946176	1439664	7	193.55
2	2	2048	32	32	14	1103872	1679600	8	226.32
2	2	2048	64	16	10	110080	164060	2	20.73
2	2	2048	64	16	12	132096	196868	2	25.80
2	2	2048	64	16	14	154112	229676	2	29.20
2	2	2048	64	32	10	788480	1198919	6	162.86
2	2	2048	64	32	12	946176	1438697	7	197.29
2	2	2048	64	32	14	1103872	1678478	8	227.69
2	2	2048	128	16	10	110080	161443	2	20.71
2	2	2048	128	16	12	132096	193727	2	25.28
2	2	2048	128	16	14	154112	226011	2	29.02

Software Pipeline Depth	**# Accelerators**	**BRAM Size**	**SPM Size**	**P**	**Depth**	**# Nodes**	**Schedule Length**	**Clusters**	**Runtime (s)**
2	2	2048	128	32	10	788480	1188278	6	159.17
2	2	2048	128	32	12	946176	1425916	7	193.36
2	2	2048	128	32	14	1103872	1663582	8	226.88