A Methodology for Hardware-Software Codesign

by

Myron King

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 9, 2013

Certified by .....
Arvind
Johnson Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

A Methodology for Hardware-Software Codesign

by

Myron King

Submitted to the Department of Electrical Engineering and Computer Science
on August 9, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Special purpose hardware is vital to embedded systems as it can simultaneously improve performance while reducing power consumption. The integration of special purpose hardware into applications running in software is difficult for a number of reasons. Some of the difficulty is due to the difference between the models used to program hardware and software, but great effort is also required to coordinate the simultaneous execution of the application running on the microprocessor with the accelerated kernel(s) running in hardware. To further compound the problem, current design methodologies for embedded applications require an early determination of the design partitioning which allows hardware and software to be developed simultaneously, each adhering to a rigid interface contract. This approach is problematic because often a good hardware-software decomposition is not known until deep into the design process. Fixed interfaces and the burden of reimplementation prevent the migration of functionality motivated by repartitioning.

This thesis presents a two-part solution to the integration of special purpose hardware into applications running in software. The first part addresses the problem of generating infrastructure for hardware-accelerated applications. We present a methodology in which the application is represented as a dataflow graph and the computation at each node is specified for execution either in software or as specialized hardware using the programmer's language of choice. An interface compiler as been implemented which takes as input the FIFO edges of the graph and generates code to connect all the different parts of the program, including those which communicate across the hardware/software boundary. This methodology, which we demonstrate on an FPGA platform, enables programmers to effectively exploit hardware acceleration without ever leaving the application space.

The second part of this thesis presents an implementation of the Bluespec Codesign Language (BCL) to address the difficulty of experimenting with hardware/software partitioning alternatives. Based on guarded atomic actions, BCL can be used to specify both hardware and low-level software. Based on Bluespec SystemVerilog (BSV) for which a hardware compiler by Bluespec Inc. is commercially available, BCL has been augmented with extensions to support more efficient software generation. In BCL, the programmer specifies the entire design, including the partitioning, allowing the compiler to synthesize efficient software and hardware, along with transactors for communication between the

partitions. The benefit of using a single language to express the entire design is that a programmer can easily experiment with many different hardware/software decompositions without needing to re-write the application code. Used together, the BCL and interface compilers represent a comprehensive solution to the task of integrating specialized hardware into an application.

Thesis Supervisor: Arvind
Title: Johnson Professor of Electrical Engineering and Computer Science

Acknowledgments

I must first acknowledge the guidance and support of my advisor Arvind. If I have matured at all as a researcher during my career at MIT, I have him to thank. John Ankcorn and Jamey Hicks have also given me excellent advice over the years and I am grateful to them too.

The contributions of my colleagues in CSG are also noteworthy. Nirav Dave and I collaborated closely on the conception and implementation of BCL. Asif Khan, Abhinav Agarwal, and Oriol Arcas were instrumental in bringing up new FPGA platforms and implementing benchmarks. I have benefited greatly from working with these four spectacular engineers, as I have with all the other members of CSG.

Lastly, I need acknowledge the love and support of my wife Adriana. I couldn't have finished this without her.

This work was supported by Nokia Research, the NSF (#CCF-0541164), Quanta Computer, and a NRF grant from the Korean Government (MEST) (#R33-10095).

Contents

List of Figures

Chapter 1

Introduction

Modern mobile devices provide a large and increasing range of functionality, from high-resolution cameras, video and audio decoders, to wireless base-bands that can work with a variety of protocols. Due to power and performance considerations, much of this functionality relies on specialized hardware. The integration of special purpose hardware into applications running in software is difficult for a number of reasons. Some of the difficulty is due to the difference between the models used to program hardware and software, but great effort is also required to coordinate the simultaneous execution of the application running on the microprocessor with the accelerated kernel(s) running in hardware. This thesis is about solving this problem.

1.1 The Difficulty With Specialized Hardware

Current design methodologies for embedded applications require an early determination of the design partitioning which allows hardware and software to be developed simultaneously, each adhering to a rigid interface contract. This is done to accelerate the development process, an important consideration with time-to-market pressures.

This rigid approach to interfaces is problematic because often a good hardware-software decomposition is sometimes not known until deep into the design process. Fixed interfaces and the burden of reimplementation late in the design process prevent the migration of functionality motivated by repartitioning. The final integration of the hardware and software

components is often difficult since, in practice, the interface specification rarely matches the actual implementation. This happens because the hardware specifications are often incomplete, and either leave room for misinterpretation or are simply unimplementable. The interface is usually specified using a design document written in a natural (human) language. While it is possible to capture certain aspects of a hardware interface accurately in English, other aspects like the timing present a greater challenge. Worst of all, by prematurely restricting the design, lower-cost or higher-performance alternatives may be ignored.

When considering specialized hardware, designers often start with a pure software implementation of an algorithm written in C. Using a profiling tool, computationally intensive parts are then identified to be implemented in hardware in order to meet the design constraints. When considering how best to implement the specialized hardware, designers can choose from a number of different alternatives:

- **Completely Customized:** Specialized hardware can be designed for the task at hand, and synthesized as an ASIC or executed on FPGA. In this case the design challenge reduces to whether the source description is amenable to generation of good quality hardware, and how efficiently can the hardware and software components be coordinated.

- **Standard IP:** The accelerator may also be available in the form of a standardized IP block, like FFT, to be invoked from the application software, though additional logic (either in hardware of software) will be required to adapt the standard block to the task at hand. The integration of existing hardware IP into a software application can be quite disruptive to the application code structure, and the challenge of generating the coordinating infrastructure remains unchanged.

- **DSP Solutions:** Lastly, the accelerator could be in the form of a programmable processor like a DSP with its own tool chain and distinct programming model. The integration and coordinating challenges persist for this case as well.

Regardless of what kind of accelerator is eventually used, some software is always required to drive it, and in all three cases, the designer will need to modify some parts

of the application code to make use of the accelerator. The efficiency of this additional software is crucial to the overall performance of the system.

Additional hardware will also be required to connect the specialized hardware to the communication fabric. Once the software and hardware components have been specified, the task of implementing the communication over a shared communication fabric can present as great a challenge as the application-level tasks, since knowledge of low-level communication primitives can be hard-won.

This thesis presents a two-part solution to the integration of special-purpose hardware into applications running in software. The first part addresses the problem of organizing the application components and generating infrastructure for hardware-accelerated applications. We present a methodology for hardware-software codesign in which the application is represented as a dataflow graph and the computation at each node is specified for execution either in software or as specialized hardware using the programmer's language of choice. An interface compiler as been implemented which takes as input the FIFO edges of the graph and generates code to connect all the different parts of the program, including those which communicate across the hardware-software boundary. This methodology, which we demonstrate an FPGA platform, enables programmers to effectively exploit hardware acceleration without ever leaving the application space.

With the application organized into a dataflow graph, the programmer has the freedom to implement the computation at each node in his language of choice. While the interface compiler can be used to re-generate the hardware-software interface, moving functionality across the hardware-software boundary will still necessitate the reimplementation of the node, unless the functionality was specified in a manner amenable to both hardware and software generation.

The second part of this thesis presents an implementation of the Bluespec Codesign Language (BCL) to addresses the difficulty of experimenting with hardware-software partitioning alternatives. Based on guarded atomic actions, BCL can be used to specify both hardware and low-level software. BCL is a extension of Bluespec SystemVerilog (BSV) [10], a commercial language for hardware synthesis, and has been augmented with extensions to support more efficient software generation. In BCL, the programmer specifies the entire de-

sign, including the partitioning, allowing the compiler to synthesize efficient software and hardware, along with transactors for communication between the partitions. The benefit of using a single language to express the entire design is that a programmer can easily experiment with many different hardware-software decompositions without needing to re-write the application code.

In [44], we demonstrated the use of Bluespec Codesign Language (BCL) to define the entire dataflow graph, including the functionality at each node. This can be a convenient option when building an application from whole-cloth, but it can also be burdensome since it requires implementing the entire algorithm in BCL. The methodology described in the first part of this thesis provides an incremental approach which allows the programmer to exploit existing IP (specified in C++, Verilog, or VHDL) inside a dataflow graph whose structure is specified using BCL. Because dataflow graphs are hierarchically composable, any node can itself describe a subgraph specified entirely in BCL whose partitioning can be further explored.

1.2 Thesis Contributions

Some of the material presented in Chapters 4 and 5 represents collaborative work undertaken with Nirav Dave, which was presented in his PhD Thesis. Where this is the case, it will be noted explicitly. The contributions made in this thesis are listed below:

1. **A Programming Model for Hardware-Software Communication:** We propose a programming model for use when integrating specialized hardware, in which all communication between hardware and software partitions is expressed using explicit dataflow. The hardware-software interface can then be inferred simply as the union of all the dataflow edges crossing the hardware-software boundary. This programming model distinguishes itself from other commonly used models for HW-SW communication by significantly raising the level of abstraction; the scheduling and granularity of the underlying implementation is left completely unspecified. An interface compiler has been implemented which generates code that can efficiently leverage the underlying communication fabric to implement the hardware-software interface and

coordinate the simultaneous execution of the application components. We show that automatically generating the transaction scheduler and burst accumulator does not negatively impact the performance of the applications in the benchmark suite, when compared to a manually scheduled alternative.

2. **An Efficient implementation of the Bluespec Codesign Language:** BCL has been implemented as an embedded domain-specific language in Haskell. Using BCL, a programmer can specify an entire program and use the type system to select a safe hardware-software decomposition. We have implemented a compiler which generates Bluespec SystemVerilog (BSV) specifications of the hardware partitions, C++ implementations of the software partitions, and a high-level dataflow description of the program to be used when mapping the synthesized code to a physical platform. The detailed contributions of this implementation are as follows:

   - **Application of Rule-Based Specification to HW-SW Codesign:** The automatic synthesis of systems specified using Guarded Atomic Actions (Rules) to a combination of hardware and software is a novel contribution. The semantics of BCL which relies the *self-failure* property of rules selected non-deterministically for execution lies at the heart of this novelty.
   - **Efficient SW generation from rules:** We present a comprehensive strategy for the generation of efficient rule implementations in software based on (but not limited to) the sequentialization of actions composed in parallel within a rule, and a set of algebraic transformations known as *when lifting* which have the effect of computing the weakest precondition of a guarded action without adding any computation to the rule body. The use of aggressive static analysis to generate efficient SW from a non-deterministic transactional system replaces traditional techniques relying on eager scheduling or run-time support.
   - **Program partitioning using the type system:** We provided the first implementation of type driven program partitioning. By integrating computational domains into interface types, programmers can implement powerful library modules which are polymorphic in both the algebraic and domain type.

3. **A SMT-driven Algorithm for Scheduling Bluespec Rules in Software:** The final contribution of this thesis is the formulation and implementation of an algorithm for the generation of efficient software rule schedules. The algorithm relies on the use of an SMT solver to construct and refine an abstract state transition graph corresponding to the BCL program under consideration. If the algorithm converges, the resulting schedule will minimize the number of dynamic guard failures encountered when executing the program. The non-deterministic execution semantics of Bluespec allow us to aggressively prune the abstract state transition graph to reflect a high-level scheduling strategy. This gives us an advantage over traditional model-checking approaches which retain all transitions, resulting in much larger graphs.

1.3 Thesis Organization

We begin in Chapter 2 by examining a number of different applications running on an FPGA platform and what effect the introduction of specialized hardware has on their throughput and power consumption. In Chapter 3 we introduce the methodology for introducing specialized hardware into applications running in software, and show how the communication FIFOs can be automatically synthesized on supported platforms. This concludes the first part of the thesis. The second part of the thesis consists of Chapters 4 and 5, where we present the Bluespec Codesign Language and explain how it is compiled to efficient software. The third and final part of the thesis is contained in Chapter 6, where we present an SMT-driven algorithm for rule scheduling in software. We discuss related works in Chapter 7 and conclude with Chapter 8.

Chapter 2

Accelerating Applications Using Specialized Hardware

2.1 Why FPGAs Sometimes Outperform Microprocessors

In Chapter 1, we made numerous references to the "Specialized Hardware". For the remainder of this thesis, we will assume that all specialized hardware is implemented in FPGA. Of course, there are often substantial power/performance advantages to implementing specialized hardware in ASIC over FPGA, but doing so requires large economies of scale and substantial experience in the ASIC tool chain, neither of which is possessed by the author. Nevertheless, much of the analysis and many of the arguments hold across the two substrates. Moreover, as the efficiency and density of FPGA improves we believe they will be used increasingly as a compute platform and not just for prototyping.

Invented in the 1980s, Field Programmable Gate Arrays, or FPGAs, consist of large numbers of reconfigurable computing elements implemented using LUTs. These LUTs are integrated with additional buffers and some interface logic into Slices, which are are replicated in a regular patter to form the FPGA fabric. Within the fabric, many FPGAs will have additional integrated ASIC blocks such as DSP slices, small distributed memories known as block-RAMs (BRAMs), and sometimes even microprocessor cores. Finally, the fabric is connected to a larger memory store, usually through a standard bus protocol. By reconfiguring the LUTs, a programmer can simulate any circuit within the resource constraints

of the FPGA fabric.

In certain domain spaces, FPGA implementations have been shown to achieve substantial speedups over their software counterparts [13,61,66] in spite of the relatively low clock speeds. Qualitatively, the reasons for these speedups are clear. FPGAs expose parallelism at all levels: bit, instruction, loop, and task. Whereas a Von-Neumann architecture must fetch an instruction from memory before executing it and subsequently store the result, an FPGA has a custom data-path which can avoid the hazards of fetching instructions and is amenable to pipelining and replication. Finally, FPGAs have specialized memory interfaces which can often be used to great advantage. Detailed studies have been performed in an attempt to more precisely identify the source of the speedups [54]. This work provides a good background to understanding the results presented in this chapter.

2.2 Comparing Partitioning Choices

For any application, a particular HW-SW partitioning choice can be evaluated on the basis of both performance (throughput) and energy. In many cases, improved performance implies lower energy consumption: if the same amount of work can be accomplished in less time, the hardware can be powered off, thereby consuming no energy at all. By this logic, moving functionality to specialized hardware is always desirable since a dedicated data-path will always be faster than software and consume less energy for a given task [54]. However, there are also cases where additional hardware can increase energy consumption. These are usually due to a mismatch between HW allocation and memory bandwidth, and must be considered as well.

The most common motivation to partition an application is that it is often not possible (or at least impractical) to implement the entire application in HW due to resource constraints. In a world of limited resources we must choose carefully which blocks are moved to hardware in order to optimize the ratio of resource consumption per unit energy savings. Consider the scenario where multiple applications are being run simultaneously by an operating system. Each of these applications might benefit from specialized hardware, and because the available FPGA resources have a firm limit, resource usage becomes an hard

constraint. Under these conditions, it is possible that a partial HW implementation of each application competing for resources might maximize overall system throughput. This is a high-level decision to be made in the operating system, and the analysis presented in this chapter demonstrates how to inform this decision.

Moving only part(s) of an application to HW will not always improve the application throughput. On the Zynq platform, for example, HW implemented on the FPGA fabric has a slower clock speed compared to the ARM core. Moving a component into HW will improve performance if considerable HW parallelism can be exploited, or the overheads associated with SW can be diminished. Partial implementation in HW invariably requires dataflow across the HW-SW boundary, and the latency resulting from this communication must also be compensated for. Depending on how the application software is organized, the synchronization required to move data back and forth between software might introduce delays by idling both the SW and HW for long periods of time. Even when the communication latency can be hidden (usually through pipelining), the cost of moving data may still overshadow any performance improvements with the particular module which has been moved from software to hardware.

Two ways of partitioning the same application can always be compared based on their throughput, energy consumption, or resource usage in isolation. In situations such as the one described earlier where an operating system needs to arbitrate between competing applications, a more complex analysis is needed which takes multiple aspects of the design into account. For this case study, we have come up with a metric we call the Energy-Delay-Area Product, by which we can compare two different HW-SW partitionings. This is just an example of a metric which could then be used by the operating system to divide resources between competing applications:

Energy-Delay-Area Product: To evaluate the "quality of results" for each application partition, we use a figure of merit (FOM) computed as the product of the following terms:

- Total energy consumed in executing the benchmark partitions, computed as the sum of SW and HW (including communication) energy components. We assume that when a component becomes inactive, its power supply can be shut-off.

- Delay computed as the total execution time required to execute the benchmark, measured using a free running clock.
- Area computed as the sum of percentage utilization of computational resources – FPGA resources include registers, LUTs, BRAMs and DSP slices, and SW resource is the application processor.

2.3 Case Studies

To motivate this thesis, we have selected a set of realistic benchmarks representing applications that are widely used in embedded systems. In the following discussion, we present an analysis of the benchmarks and an evaluation of the various partitioning choices for their implementation executed on the Zynq platform from Xilinx. In Figure 2-1, we show the speedup obtained by each HW-SW partitioning over the full SW implementation (partition A) for each benchmark. Vorbis has six partitionings (labeled A to F), while the other benchmarks have four (labeled A to D). Their details will be described later in the section.

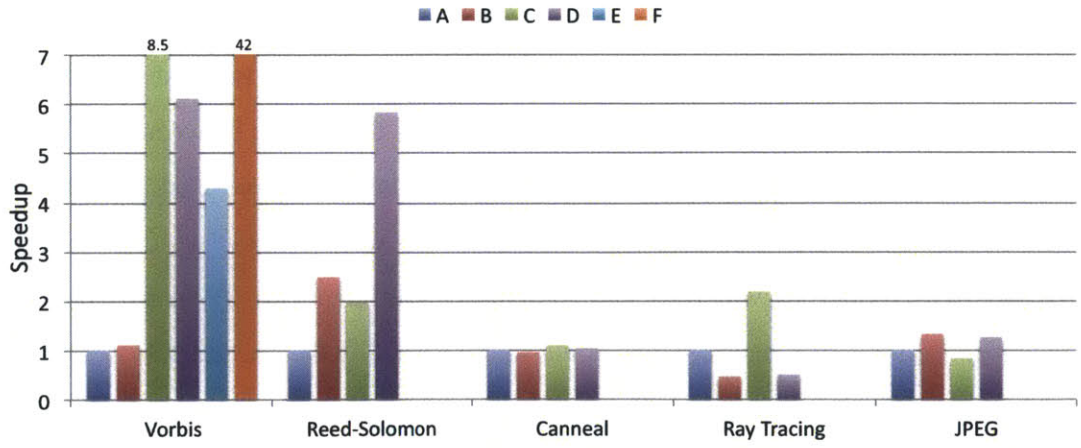


Figure 2-1: Speedup obtained by the various HW-SW partitions over the full SW (A) partition (higher is better)

Figure 2-2 provides a breakdown of the execution time for each partition into the SW, HW and communication times. Since these components can run concurrently, the total execution (wall clock) time can be less than the sum of the three time components. The

total execution time is indicated by a black bar in each column. In Figure 2-3, we show, for each benchmark, the energy consumption of the partitions, normalized to the full SW partition (A). Figure 2-4 provides the FOMs for the various partitioning choices of the benchmarks.

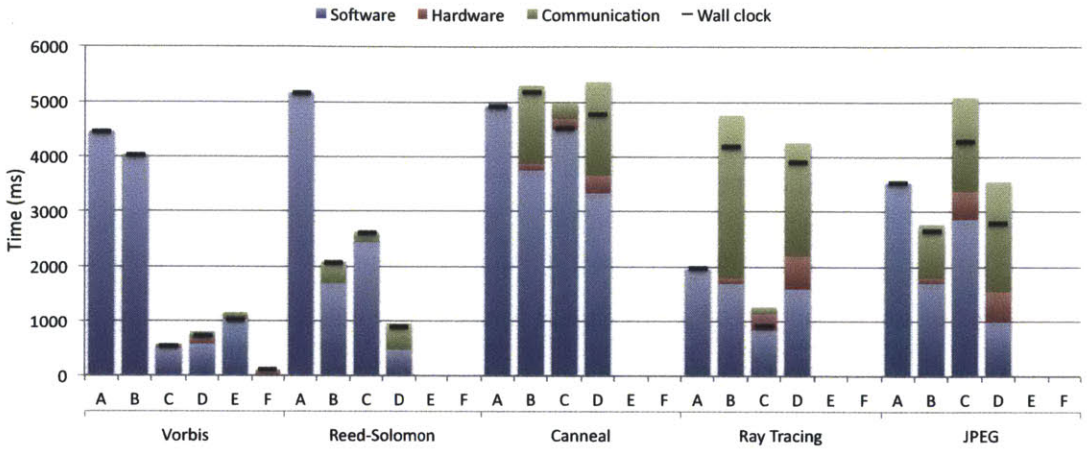


Figure 2-2: SW, HW and communication times for the various application partitions. The total execution time is indicated by a horizontal black bar in each column, which may be less than the sum of the three time components due to concurrent execution.

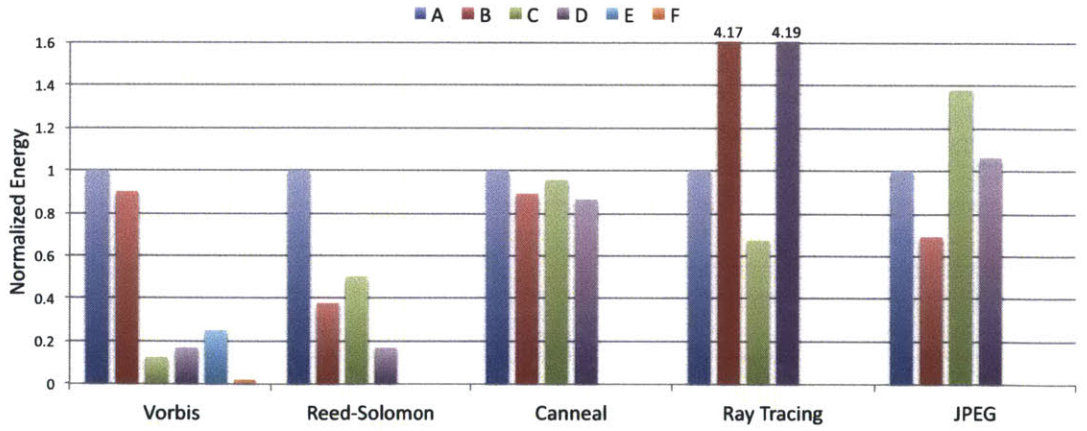


Figure 2-3: Energy consumption of the various HW-SW partitions normalized to the full SW (A) partition (lower is better)

Benchmark	A	B	C	D	E	F
Vorbis	4.96	4.44	0.10	0.17	0.34	**0.004**
Reed-Solomon	6.66	1.07	1.90	**0.22**	-	-
Canneal	6.06	6.11	5.76	**5.57**	-	-
Ray Tracing	0.96	19.0	**0.49**	20.5	-	-
JPEG	**3.12**	3.24	6.71	6.18	-	-

Figure 2-4: Figure of merit (energy-delay-area product) comparison for the various application partitions (lower is better)

2.3.1 Ogg Vorbis Decoding

Ogg Vorbis [2] is an audio compression format aimed at low-complexity decoding. The front-end of the Vorbis decoder pipeline consists of a stream parser and various decoders through which the audio frequency spectra are reconstructed. The back-end of the pipeline transforms the signal using an IMDCT module which internally uses the computationally intensive IFFT. The frames in the output are overlapped using a sliding window function to compensate for spectral leakage at the boundaries. Figure 2-5 shows the dataflow and the HW-SW partitions chosen for analysis.

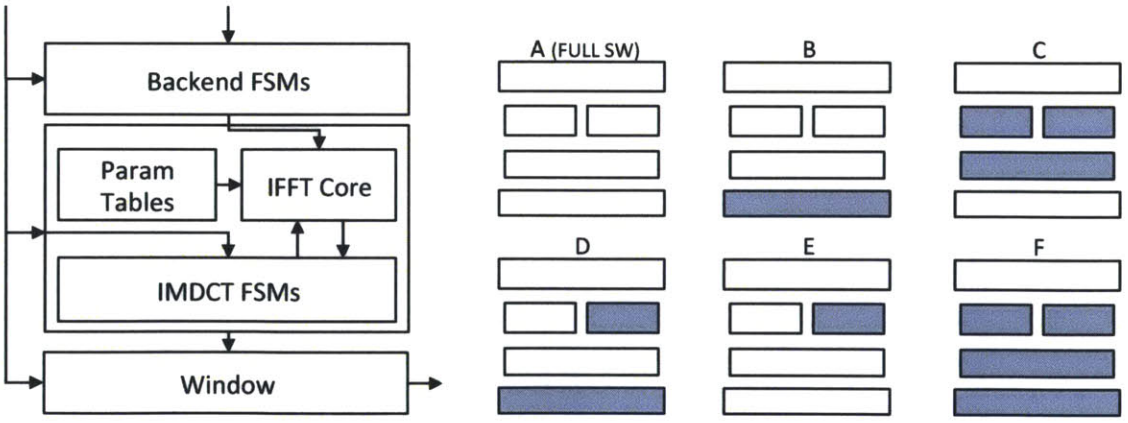


Figure 2-5: Vorbis dataflow and partitions (HW modules are shaded)

Our first attempt at accelerating Vorbis is moving the windowing function into HW (partition B). Though the windowing function has a very efficient hardware implementation, the relative weight when compared to the software implementation is not enough to compensate for the incurred cost of communication. As a result, the speedup in partition B

is negligible.

Moving the IMDCT functionality to HW in partition C results in a large speedup, as shown in Figure 2-1, because IMDCT is able to exploit significant parallelism through a very efficient IFFT used to implement its core functionality. Though the dataflow across the HW-SW boundary for partition C is significant, the speedup in compute is able to compensate for some of the added latency. Additional latency is also hidden by transferring entire frames in large bursts from SW to HW and back again. Moving the Windowing function to hardware in partition F further improves the throughput of partition C. The same amount of data is transferred across the HW-SW boundary which lets observe speed difference between HW and SW implementations of the windowing function.

Though the IFFT core of IMDCT is in HW for both partition D and E, they are significantly slower than C. Given the overwhelming weight of IFFT in the computation of IMDCT, this is surprising until we consider the fact that a single IMDCT will invoke IFFT repeatedly, incurring a significant penalty for all the communication between HW and SW.

All the partitioning choices are able to achieve relatively good HW-SW concurrency. This is reflected in Figure 2-2 where the black line on entry for each Vorbis partitioning is well below the top of the shaded region, indicating that the total execution time is less than the sum of the active HW and SW components.

As shown in Figure 2-3, partition F achieves the least normalized energy consumption. The combination of the speedup and energy efficiency, coupled with a moderate area cost, allows F to achieve the best FOM (0.004), shown in Figure 2-4, significantly smaller than other partitions. This is a result of the highly efficient and parallel HW implementation of the complex and large IMDCT computation.

A final comment on this benchmark is that due to the extremely efficient implementations of blocks moved to the FPGA, the red component corresponding to "time spent executing application functionality in hardware" is hardly visible in Figure 2-2 compared to the communication and application SW time components. The reasons for this performance improvement were discussed briefly in Section 2.1, and will be demonstrated by some of the subsequent benchmarks as well.

2.3.2 Ray Tracing

Ray tracing [55] begins by loading the scene geometry into memory, after which the BVH Ctor module performs an initial pass to construct a bounding volume hierarchy (BVH). RayGen constitutes the first step of the computation, generating rays that originate at the camera position and pass through each point in the field of view at a specified granularity. BVHTrav implements the core of the ray tracing pipeline. As the algorithm traces a ray through the scene, it traverses the BVH, testing for intersections with the sub-volumes, and finally with the geometry primitives at the leaves of the BVH. Figure 2-6 shows the dataflow and the four partitions considered in this study.

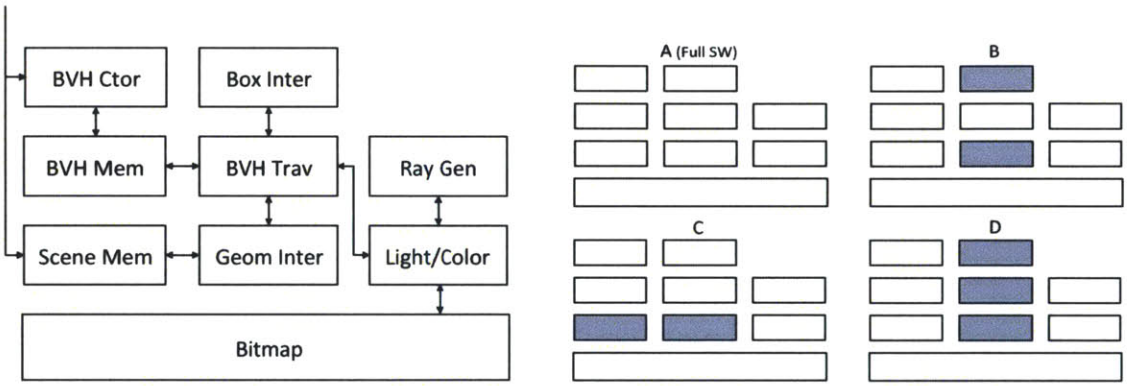


Figure 2-6: Ray tracing dataflow and partitions (HW modules are shaded)

The most intensive computation in Ray tracing is calculating collisions between the scene geometry and the rays, and the intersection calculations required to traverse the BVH. When implemented in HW, these computations can make effective use of the large number of DSP slices available on the FPGA when combined to form deeply pipelined structures which are capable of outperforming the CPU and producing one result per cycle. The challenge in accelerating ray tracing is to effectively utilize these deep arithmetic pipelines. Within the context of a single ray, parallelization (replicating the collision pipelines) will provide little advantage. When traversing the BVH, successive calculations are dependent so algorithmically there is no need to have multiple comparisons in flight at the same time. Pipeline replication might be useful when computing collisions between a ray and the geometry at the leaf node of the BVH since each primitive needs to be tested independently. On the other hand, if the BVH tree has been effectively constructed, the number of primi-

tives at each leaf node will be low. In this benchmark, this number was lower than the depth of the collision pipeline. This fact, along with resource constraints, prohibited replication of the collision pipeline. Of course, each ray can be computed independently, so at the highest level this application has ample parallelism which can be exploited with sufficient FPGA resources.

Partition B moves the collision computations into HW. However, this leads to a slowdown, as seen in Figure 2-1, as the BVH traversal generates non-uniform memory accesses preventing efficient bursty communications and requiring additional synchronization. Not only does this non-uniform access patter prevent us from transmitting data between HW and SW in large bursts, it also prevents us from utilizing the collision pipelines fully. This slowdown should come as no surprise; this type of memory access is always a challenge in high-performance computing. Traditionally caches are very effective in improving performance in these cases, especially if the working set is small enough. Without a caching infrastructure (as is the case with this application), we will pay a heavy price.

Partition C relies on the observation that once we reach a leaf node, we know the range of geometry we need to examine. A single burst command transmits the necessary geometry to HW and the collisions are calculated by the pipelined HW, allowing C to have the maximum speedup and the best FOM (0.49), as seen in Figure 2-4. IF we spent more time optimizing the software, it is conceivable that this partitioning could see substantially greater speedups. In particular, support for multi-threading would allow us to multiply this single-thread performance improvement by the number of concurrent threads. Finally, attempting to put the traversal engine into HW in partition D introduces similar synchronization hazards as B, just in the opposite direction. Both B and D have a very poor FOM due to the slowdown and extra resource utilization.

In Figure 2-2, we can see that the HW-SW concurrency we were able to achieve for this application varies greatly across partitioning choices. Partitions B and D are particularly poor, due to the synchronization required interleaving HW and SW to traverse the bounded volume hierarchy. Partition C, the only partition to achieve a speedup, was able to achieve concurrent execution of HW and SW.

2.3.3 Canneal

A member of the PARSEC benchmark suite [15], Canneal performs simulated annealing to optimize the routing cost of chip design. The dataflow of the simulated annealing algorithm is shown in Figure 2-7. It repeatedly selects two random elements from the supplied net-list, calculates the change in routing cost resulting from swapping their locations, and decides whether to perform the swap, until the termination criterion is met. In the end, it calculates the total routing cost.

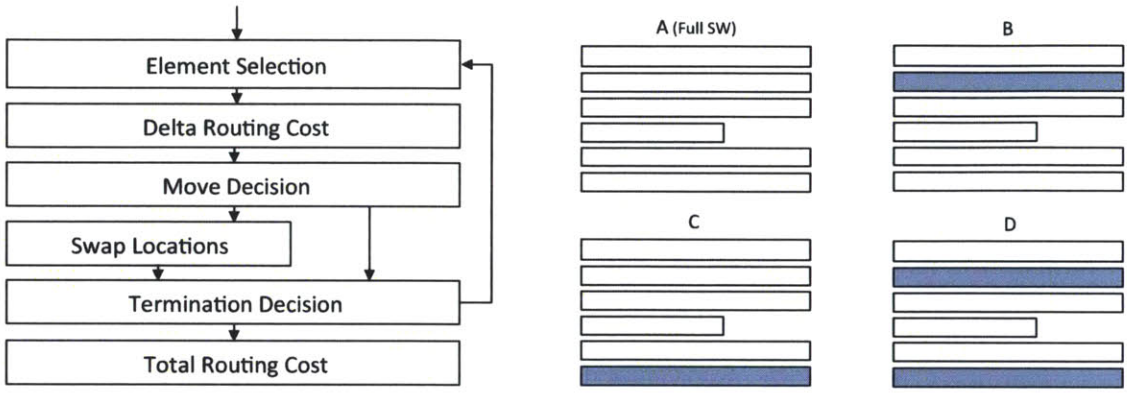


Figure 2-7: Canneal dataflow and partitions (HW modules are shaded)

This benchmark is interesting because none of the partitioning choices were able to achieve a meaningful speedup over pure SW. This is primarily because the state of the SW organization did not permit us to expose any parallelism in the problem. Algorithms like this are normally parallelized by allowing multiple worker threads to operate on disjoint regions of the graph. SW Synchronization primitives are inserted to ensure consistency between concurrent threads. This single-threaded implementation included no such optimizations, so the HW was used only to improve the single thread performance. As such, it was difficult to hide the communication latency and keep the hardware busy. It is likely that multi-threading the SW would enable a HW-SW decomposition with substantially improved throughput.

Moving Delta Routing Cost from SW to HW in partition B results in a slight slowdown, as shown in Figure 2-1. The HW parallelism in Delta Routing Cost is not enough to make up for the substantial overhead of HW-SW communication, as shown in Figure 2-2.

On the other hand, moving Total Routing Cost to HW in partition C results in improved performance because of the overlap in hardware, software, and communication. Partition C, however, has higher energy consumption than partition B because of the longer SW execution time. Partition D, where both Delta Routing Cost and Total Routing Cost are implemented in HW, results in a moderate speedup, the lowest energy consumption, and the best FOM.

2.3.4 JPEG

This benchmark is a decoder for files encoded using the JPEG specification [62]. The input for the implementation are JFIF [38] files, which include the information needed to decode a JPEG image, such as Huffman and quantization tables. The file headers specify the size of the image, the number and resolution of components, and the algorithm used for entropy coding. Figure 2-8 shows the dataflow of the decoding process, and the design partitionings implemented for this benchmark.

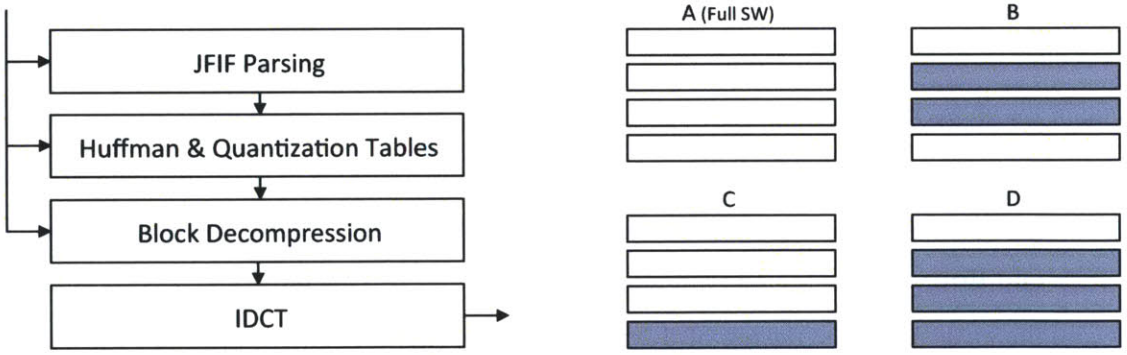


Figure 2-8: JPEG Decoder partitions (HW modules are shaded)

The reading and parsing of the JFIF files is always done in SW. Partitions B and C are complementary: B does the first half, decompression and entropy decoding, while C does the second half, IDCT, in HW. Partition D has all the decoding in HW. The results in Figure 2-1 show that B achieves a speedup, C has a slowdown, while D performs as well as software. Someone familiar with JPEG would expect partition C to outperform software, and while it is possible that the IDCT is sub-optimal, the reason that this causes a slowdown is that the IDCT is not large enough. If it were larger, the speedup from moving it to

HW would outweigh the cost of moving frames back and forth between HW and SW.

Figure 2-2 shows that communication time is the main culprit. HW-SW communication in partition B consists of only 2.2 MB of data, which results in a very short transmission time and reduced energy consumption (Figure 2-3). In contrast, the large amount of data consumed and produced by C (6.2 MB) and D (4.5 MB) increases their communication latencies and energy consumption. Even though partition B has higher performance and lower energy than the SW-only partition (A), Figure 2-4 shows that the latter still obtains the best FOM (3.12), followed closely by B (3.24). The moderate performance, high energy consumption and high resource usage of C and D raise their FOM to 6.71 and 6.18, respectively.

2.3.5 Reed-Solomon

This benchmark is a Reed-Solomon decoder, an efficient and customizable error correction code [67], for an 802.16 receiver. The algorithm's dataflow and the HW-SW partitions chosen for characterization are shown in Figure 2-9. Interaction with the outside world occurs through the IO data transfer module. This block reads the packet header to obtain the dynamic control parameters, lengths of the data n and the parity t blocks. It then transmits these parameters to the computation modules before supplying the data for every frame. There are five computation modules: Syndrome, Berlekamp-Massey, Chien, Forney and Error Corrector. The amount of computation is dynamically determined by n, t and the error rate. The design of this application is based on an efficient hardware implementation [7]. Reed-Solomon is a protocol which was designed for HW implementations, so we expect that moving any block to HW will improve application throughput if it is implemented correctly.

The data transfer and control setup block is always in SW. Moving any block to HW leads to a large speedup due to the highly efficient parallel implementation of the Galois Field arithmetic blocks used in this algorithm. In B, we implemented the first two modules in SW, which includes the computationally intensive Berlekamp-Massey block, and the remainder in HW. In C, we moved Berlekamp into HW but the final error correction step

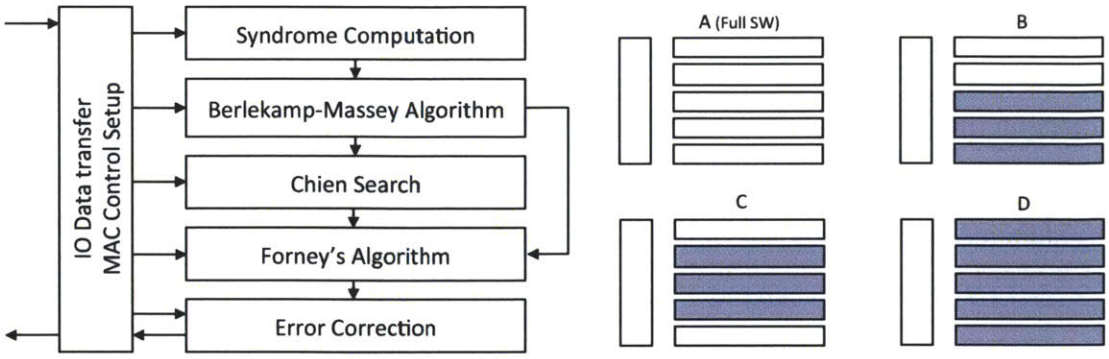


Figure 2-9: Reed-Solomon dataflow and partitions (HW modules are shaded)

is moved back to SW. As shown in Figure 2-1, partition B of Reed-Solomon has a higher speedup than C as it avoids the dependency of the final step being in SW.

The active HW duration for all partitions of this benchmark is much smaller than the active SW and communication durations, as seen in Figure 2-2. It is interesting to note that the efficiency improvements of implementing Galois Field arithmetic in FPGA is so dramatic that all speedups were obtained with minimal HW-SW concurrency. Partition D has the maximum speedup as it has all of the computation modules in HW. Partition D also has the least normalized energy consumption, seen in Figure 2-3, allowing it to achieve the best FOM (0.22), seen in Figure 2-4.

2.3.6 Evaluating Favorable HW-SW Partitionings

There are some interesting comparison points in the characterization of the different benchmarks partitions. When we compare Vorbis and JPEG, we see that both of them have a module for computing IMDCT. However, moving this module to HW results in different speedups. Partitions C and F of Vorbis have an extremely large speedup, while partition D of JPEG has a relatively small speedup. This is due to the subtle fact that the IMDCT size in Vorbis is many times larger than in JPEG. In case of Vorbis, Reed-Solomon and Ray Tracing, partitions that have the largest speedups also have the least energy consumptions. However, this is not necessary, as Canneal's partition B does not have a speedup over the full SW partition, but still has a lower energy consumption. JPEG presents the interesting case of the full SW partition having the best FOM, as none of its HW-SW partitions is able

to achieve a speedup or energy efficiency to compensate for the additional area cost.

As to what general conclusions can be drawn from the benchmarks presented in this section, the answer is not straightforward. We can certainly see trends: certain types of functionality like IFFT are ideally suited for HW execution as long as we can hide the communication latency, and applications requiring non-uniform [dependent] memory accesses (NUMA) are much more difficult to accelerate using specialized hardware. What is more difficult to ascertain is whether the results are a fundamental characteristic of the algorithm, these particular implementations, or of the Zynq platform on which the evaluation takes place. For example, if we re-executed these benchmarks on a platform with significantly different processor/FPGA/memory-interface would the optimal partitioning choice change for some or all of the benchmarks? Our answer to this question is the motivation for this thesis, by enabling programmers to more readily execute programs on these systems they can simply run them and find out.

2.4 Tool Chain and Execution Environment

Figures 2-11 and 2-10 outlines the major components of the co-development toolchain used to compile and instrument programs for this case study. When the application is specified using the Bluespec Codesign Language, developers manually specify a partitioning choice in the source code, and the rest of the flow is push-button. If a hybrid approach is being used, developers can manually generate parts or all of the C++ and BSV input.

The SW components are specified in C++ and compiled along with libraries which implement the communication APIs and runtime layer using g++. If the target is simulation, a stand-alone executable will be generated, otherwise the toolchain generates an .elf file which is downloaded to the application processor on the Zynq platform. The communication primitives available on this platform vary in bandwidth and latency. The choice of which HW communication device to use can either be specified by the programmer or left to the interface compiler, which selects the best primitive based on both the application requirements and the platform constraints. This choice is static, so the toolchain can further optimize the SW stack based on the selection.

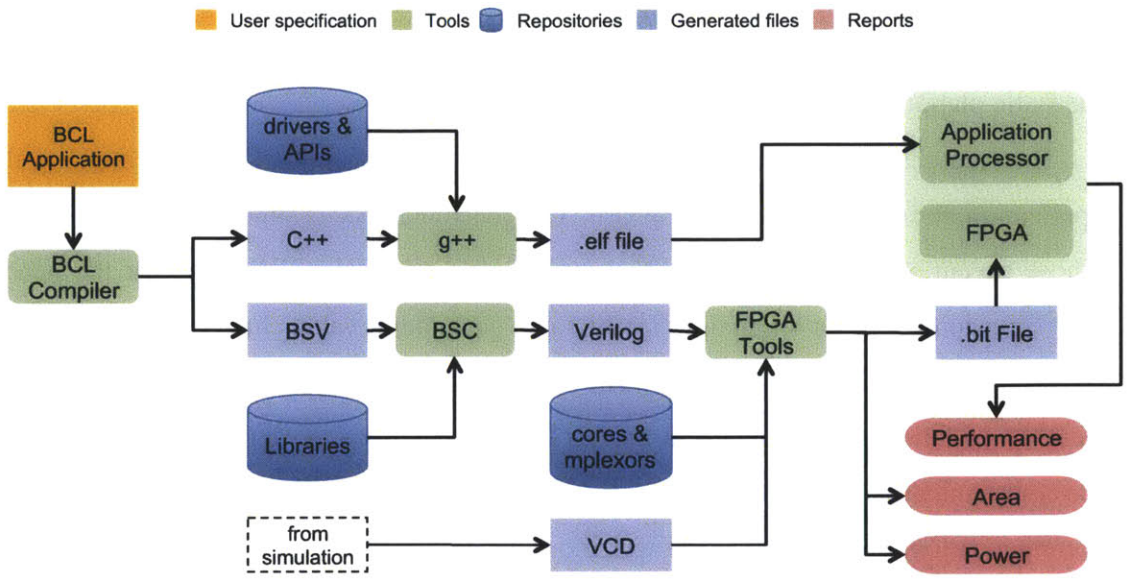


Figure 2-10: Application co-development toolchain targeting the Zynq or ML507 platforms from Xilinx. Software can be specified using BCL or C++. Hardware can be specified using BCL or BSV.

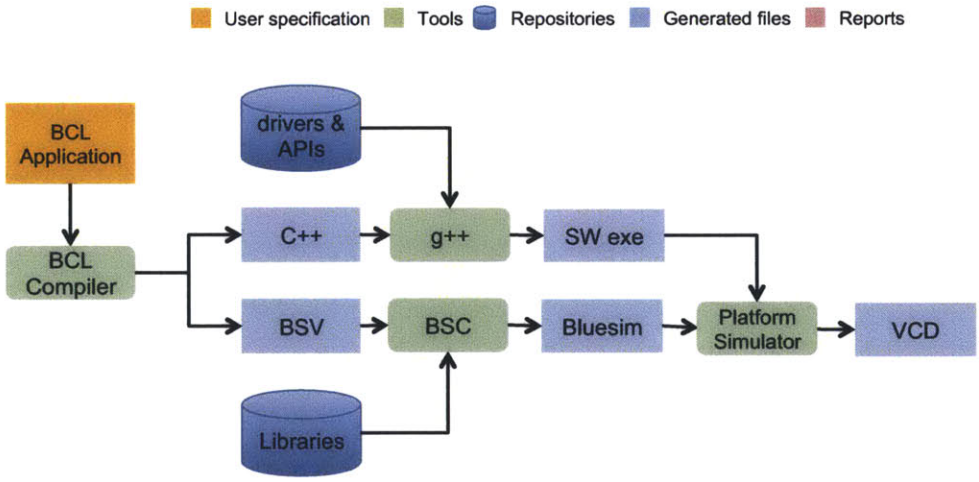


Figure 2-11: Application co-development toolchain targeting the platform simulator. Software can be specified using BCL or C++. Hardware can be specified using BCL or BSV.

On the HW side, we simulate the BSV using Bluesim (the BSV simulator) if the target is a simulation, otherwise, we compile it to Verilog using the Bluespec compiler (BSC) from Bluespec Inc. The Verilog code is then combined with the communication cores and channel multiplexors, and synthesized to a configuration file for the FPGA fabric using the

FPGA synthesis tools. Once again, the toolchain is able to make a static assignment of communication primitives and connects the assignments in the adapter layer. As with SW, the static mapping permits optimization of the full HW stack.

The tool-chain and execution environment supports co-simulation for debugging the application during the development phase. The generated BSV is simulated using Bluesim, whose execution is coordinated with a concurrently executing SW thread for the generated C++ code. The co-simulation facility also generates a value change dump (VCD) as the activity profile for measuring power. The FPGA tools provide the resource utilization statistics and the power estimates for the various computational resources. Timers built into the communication APIs and cores provide timing statistics for the software, hardware and communication time, as well as the wall clock time.

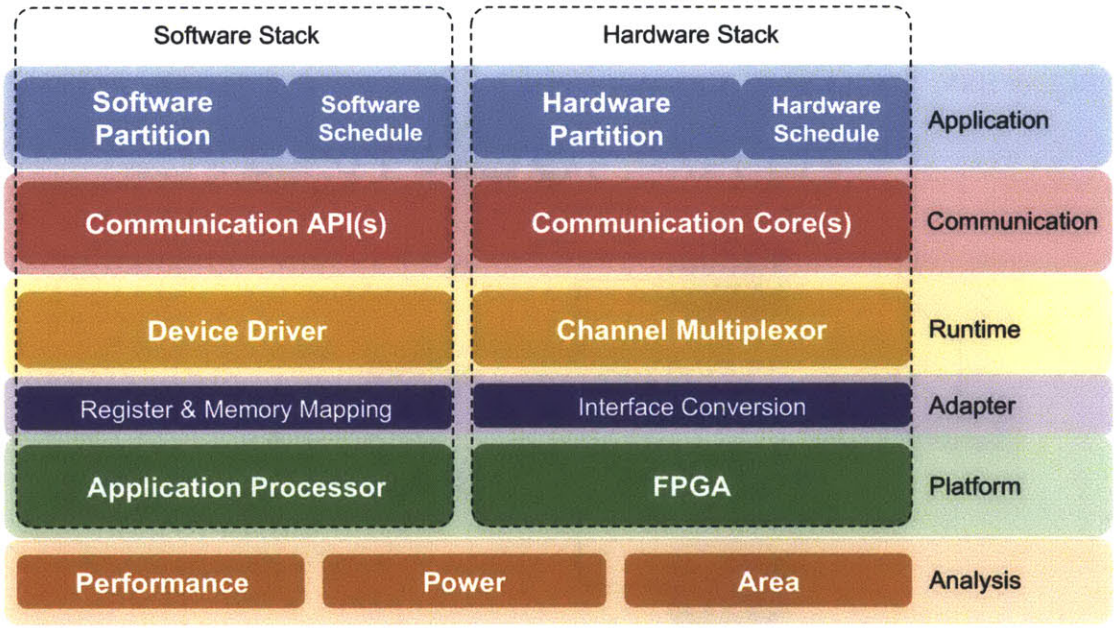


Figure 2-12: Application co-execution environment

Figure 2-12 depicts the HW and SW stacks in our co-execution environment. At the top of the stack is the application layer containing the HW and SW partitions, which communicate with each-other through language-level primitives. Dataflow across the HW-SW boundary at the source level is characterized by a set of uni-directional FIFO links. These primitives are implemented both on FPGA and in SW as the communication layer, which

provides the abstraction of named virtual channels. Below the communication layer is the runtime layer, which supports the multiplexing of physical communication channels between virtual channels.

The top three layers are platform independent and can be compiled to any supported platform. The adapter layer and below must be re-implemented for each physical platform. The runtime layer is built on an abstraction of the HW resources including memory interfaces, bus connections, etc., and each physical platform is slightly different, requiring adapters. The HW and SW stacks are compiled separately to the FPGA and microprocessor. The analysis layer is shown as the base-layer, but is actually integrated quite closely with the platform layer. This data is then combined with static analysis performed by the toolchain to characterize applications.

Our platform layer uses the Zynq device on the ZedBoard. The Zynq device contains a dual-core ARM processor connected to a 7-series FPGA fabric. The connection options range from the high-bandwidth DMA, ACP, and AXI-stream, to the low-bandwidth AXI-register. The analysis layer, shown at the bottom of the stack, will be described in the following section. Though not used in the evaluation of the benchmarks, we have reimplemented the platform layer (platform support) using the ML507 device, also from Xilinx. This platform contains a single-core PPC440 processor connected to a 5-series FPGA. The software running on the microprocessor communicates directly to the hardware running on the FPGA through the LocalLink™ interface using embedded HDMA engines.

2.4.1 Instrumenting The Benchmarks

Discerning where time is spent in each benchmark requires careful instrumentation of the application and infrastructure code. For each partitioning choice, Figure 2-2 displays the amount of time spent in HW, SW, and Communication. Instrumenting the code is more complicated than recording active periods of the HW and SW partitions, because each partition not only executes the application-level functionality, but also implements the dataflow. For each partition it was necessary to differentiate between these two aspects in order to come up with the final numbers.

Instrumenting the HW Partition: With the exception of some interface code used to interact with the peripherals connected to the FPGA fabric, the entire hardware partition of each benchmark is ultimately specified using BSV. Each rule in the partition either implements application or communication functionality, and in each module that contains application functionality we instantiate a **PulseWire** which is asserted by each application rule. Using the **ModuleState** monad, these wires are collected at the top-level and a register is incremented each time any of the wires is asserted. Following the execution of the benchmark, the register will contain the number of cycles the application functionality was active. When BCL is used to specify the hardware partition, the instrumentation described here is generated automatically.

Because the HW partition also contains the communication scheduler, there is additional functionality to record the time spent transferring data between the hardware and software partitions. Each time a transaction is initiated, the communication scheduler starts a timer and when the transaction completes, the timer is paused. Though we have the ability to record communication time spent on each virtual channel, we only report the sum of time spent on all channels.

Instrumenting the SW Partition: The ARM core has a built-in timer which we use to instrument the software partition. If C++ is implemented manually, then it is up to the developer to pause and resume the software timer. If the C++ is generated using BCL, the BCL compiler will automatically differentiate the rules implementing application functionality from those implementing communication and pause/resume the software time accordingly each time a rule has successfully fired.

2.5 The Challenges in Using FPGAs

Once the benchmark applications have been selected, the task of implementing them on an FPGA presents some unique challenges. Even familiarity with the application domain does little to ease the pain, since implementing designs on an FPGA is akin to writing software without an operating system, stdlib, or a memory hierarchy (let alone virtual memory). The problem of managing memory is usually the first to arise since an FPGA programmer

is usually provided with relatively low-level interfaces to a memory controller over some standard bus. Because there is no virtualization or cache hierarchy, all data-management within an application becomes explicit. When writing software, an important goal is to keep the size of the working set small enough to fit in cache and where this is not possible, to reduce the number of trips to memory. When creating an FPGA design, we first need to build the cache. The result is that much of the effort in making FPGA designs revolves around creating infrastructure to manage memory.

There have been attempts at providing infrastructure support to ease the use of memory by FPGA designs. Most familiar to the author is LEAP by Adler *et al.* [6], which provides a parametric memory hierarchy for a variety of FPGA-based platforms. The use of the memory hierarchy must still be explicit, since the performance implications of a cache miss (for example) are catastrophic when the backing store is in the host memory. Though library implementations of memory primitives are very useful in avoiding work, there is a more fundamental problem which do not address. If your on-FPGA storage is not sufficient to hold the working set, the performance implications of going to memory will require a drastic re-organization of the applications microarchitecture to (hopefully) hide the added memory latency.

In addition to providing infrastructure support, the challenge of FPGA portability looms large. Different devices will have different capabilities and moving an application from on to another will be difficult if dependencies cannot be met. Attempts at virtualization mentioned previously ameliorate this problem, but once again the performance implications if particular functionality is not support natively can be catastrophic.

Another example of FPGA implementation challenges are those which arise when doing arithmetic. There are certain algorithms which lend themselves to efficient pipelining until we realize that our entire design is allowed to have at most 4 64-bit multipliers and a single 64-bit fixed-point inversion operator, usually due to a limited number of DSP slices. All of a sudden the structure of a design which had hitherto assumed an arbitrary use of the "*" operator becomes almost unrecognizable. Regaining the performance after such heavy multiplexing of resources is not an easy task. Finally, when creating specialized hardware and adding it to a processor-based system, we are exposing ourselves to "real" low-level

systems bugs such as bus errors and other memory artifacts usually only known to people who write device drivers. In essence, you need to write a device driver for each piece of specialized hardware.

The most mundane, but nonetheless real problem encountered by the FPGA designer is that of the FPGA tool-chain. Even simple designs routinely take 45 minutes to an hour to synthesize. Designs which fail to synthesize often take far longer since the tools, which often take pseudo-random seeds each time they attempt place-and-route, will try for much longer before giving up. More complex designs can take multiple hours to synthesize. When a designs fails to make timing, something which often happens in the final stages of synthesis, connecting the error message to the structure in the source program is is as much an art as it is a science since designs can be optimized beyond recognition. The final insult when using FPGA tools is that they tend to be buggy: for example after weeks of searching we realized that the Xilinx tool-chain would never correctly compile a sized FIFO of length greater than three for a Virtex7-based platform, in spite of the fact that the Verilog was legal and worked in simulation.

The work presented in this thesis is primarily intended to reduce the effort required to move functionality between hardware and software, but as a side effect it does address some of the general difficulties in using FPGAs. By presenting a model for FPGA virtualization (referred to earlier as Platform Support), moving between supported FPGAs can be relatively straightforward, though the performance hazards associated with resource virtualization persist. Our tool-chains also give the programmer and end-to-end alternative to wrestling with the vendor-provided tool chains.

2.5.1 Choosing a Benchmark Suite

When compiling a software programming language, evaluating the results is often relatively straightforward. If the input language is widely used (such as C, C++ or Java), we can choose to use a standard benchmark suite. Some of the more popular benchmark suites include (but are not limited to) SPEC* [3], DaCapo [16], PARSEC [15], and EEMBC [4]. Even when the input language is non-standard, translating from one software idiom to

another is usually quite straightforward.

One major advantage to using a standardized benchmark suite is that the applications which comprise the suite require a great amount of effort to implement. Each benchmark generally represents a different problem space and has been implemented by an expert in that particular domain. The performance difference between applications written by neophytes and domain experts can be large and in some cases an inefficient implementation is of little or no use. Consider a linear algebra benchmark: the naive implementation of most matrix operations, even if they achieve an acceptable time complexity, will probably exhibit poor cache behavior. As such, even the best compiler will still generate an executable which will perform poorly due to memory bottlenecks inherent in the specification. In other words, the execution of this program will reflect not the quality of the compiled code but rather the efficiency of the memory hierarchy on which the application is being executed. Another benefit to using a standard benchmark suite is that the selection of applications represents a consensus as to which problems are important. Not only does this help language and compiler implementers concentrate on problems which their peers consider prescient, but it gives a common vocabulary when comparing techniques.

When evaluating techniques for HW-SW codesign in this thesis, we are at disadvantage due to the lack of any benchmark suites. In fact, the evaluation presented in this chapter is the only one of its kind; there are no prior works discussing even so much as the selection of benchmarks for HW-SW co-execution. Traditionally, HW-SW codesign solutions have focused on control problems where the primary concern is preserving particular latency guarantees [12, 63]. This is very different from the work presented here where performance/throughput is the primary goal. Perhaps the reason that there are no standard benchmarks for HW-SW codesign is that the suitability (or even the necessity) for HW-SW co-execution depends so much on the platform of choice. Two platforms with similar configurations but different microprocessors, for example, will almost certainly have different needs for HW acceleration since the relative costs of HW, SW, and memory bandwidth will significantly change the value of the design trade-offs. Of course, the effort of implementing the benchmark invariably follows their selection.

This long list of difficulties explains the relatively small number of benchmarks pre-

sented in this chapter. Bringing up the Xilinx's Zynq platform used to run the examples and implementing them took four PhD students an entire semester to complete. Naturally, additional benchmarks would add more depth to the explanations, but this will be left for future work.

Chapter 3

A Model For Hardware/Software Communication

In Chapter 2, we examined how adding specialized hardware affected the performance of a number of different applications. In this section, we describe a general framework which can be used when introducing HW acceleration to applications running in software. Using traditional methodologies, the task of migrating functionality from software (SW) to hardware (HW) can be characterized as follows:

- Profile the application and select the kernel to be re-implemented in HW.
- Define the interface, and implement the specialized HW.
- Translate data-types which cross the HW/SW boundary.
- Implement additional HW and SW to connect the application components to the communication fabric.
- Restructure the application SW to more effectively exploit the HW accelerator.

While some of these steps require deep insight into the organization of the algorithm, others are simply tedious and error prone. Our experience has led us to a new methodology that addresses some of the problems encountered in traditional methods. We begin by using dataflow to describe the high-level organization of each application. The computation at each node in the dataflow graph is implemented in the programmer's language of choice,

and a target for the node (either HW or SW) is also specified.

By enforcing dataflow as the only method of communication between HW and SW, we separate the task of implementing the communication into two parts. At the application level, the programmer simply defines the HW/SW interface as the union of all dataflow edges which cross the HW/SW boundary. At the systems-level, we consider the separate problem of efficiently mapping the FIFO channels to the physical platform.

With a target designated for each node in the graph, an interface compiler we have implemented specifically for this task compiles the FIFO edges and generates code to connect the nodes. Adjacent nodes assigned to SW communicate using shared memory, while adjacent nodes assigned to HW are connected using efficient FIFOs implemented in the FPGA fabric. Edges crossing the HW/SW boundary require substantially more work, but on supported platforms these too can be *automatically* implemented on the shared communication fabric, along with the SW drivers and HW shims connecting the accelerator to the rest of the system. Assuming the efficient and automatic implementation of the communication model, the primary challenge for the application programmer is now to enable the appropriate granularity of communication.

3.1 Traditional Methodologies of Integrating Specialized Hardware

In this section we describe a structured approach to the task of accelerating a C++ implementation of Ogg Vorbis running on a processor with a tightly coupled FPGA. Ogg Vorbis is an open-source psycho-acoustically-aware audio CODEC aimed at low-complexity decoding. We have chosen this example for its relative simplicity, but the complications which arise are only exaggerated as the applications grow in complexity. A high-level block diagram of the Vorbis pipeline is shown in Figure 3-2, where buffers between blocks indicate a certain degree of latency tolerance. The details of the back-end of this pipeline are shown in Figure 2-5.

The front end of the Vorbis pipeline consists of a stream parser and various decoders

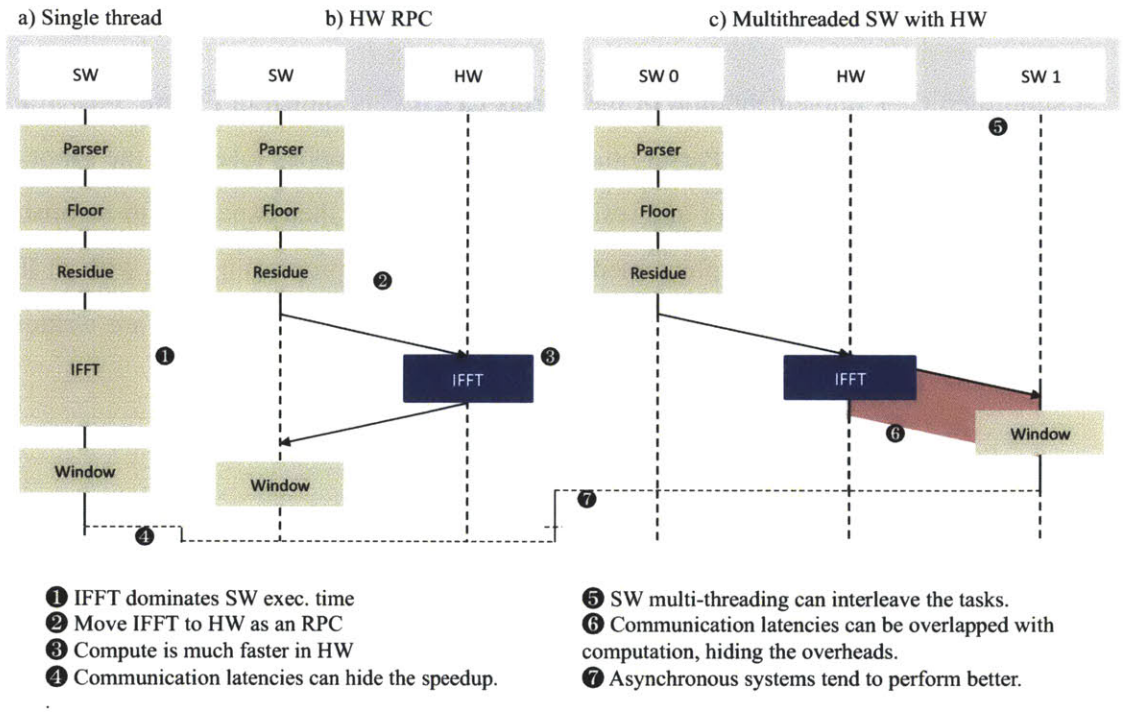


Figure 3-1: The organization of the application evolves as we introduce hardware and then optimize its performance. Markers in the following text correspond to the numbered points in this diagram.

though which the audio frequency spectra are reconstructed. Due to the compression scheme, these modules have complex control but are relatively lightweight from a computational standpoint. The back-end of the pipeline transforms the signal from the frequency to the time domain using IMDCT. The frames in the final output are overlapped using a sliding window function to compensate for spectral leakage at the frame boundaries.

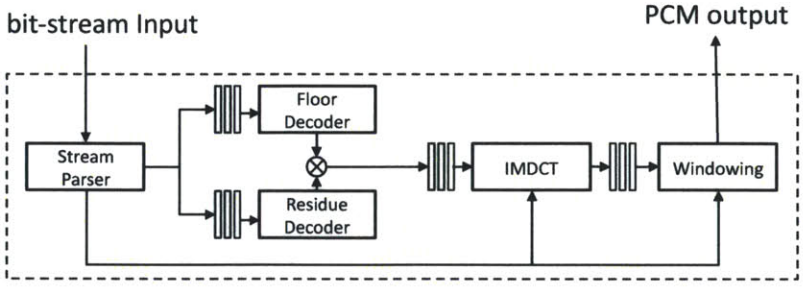


Figure 3-2: Block diagram of the Ogg Vorbis pipeline.

3.1.1 Kernel Selection

A simplified C++ implementation of the Vorbis CODEC is shown in Figure 3-3. Stream parsing is encapsulated in the Parser object, which in turn supplies tokens to the Floor and Residue constructors. The floor and residue components are then expanded to reconstruct the audio frame. IMDCT is implemented using pre- and post-processing steps with an IFFT kernel. The windowing function is applied to produce PCM packets which are written to an output device.

```
void vorbis(file* input, void* output){
 Parser p(input);
 int N = p.frameSize();
 while(!p.eof()){
   // front-end: parsing & frame reconstruction
   Floor f(p);  Residue r(p);
   Frame frame(f,r);
   int a[2*N], b[2*N], *tmp, i;
   // back-end: domain transform & windowing
   for(i = 0; i < N; i++){
     a[i]   = imdctPreLo(i,N,frame[i]);
     a[i+N] = imdctPreHi(i,N,frame[i]); }
   ifft(2*N, a, a);
   for(i = 0; i < N; i++)
     b[bitReverse(i)] = imdctPost(i,N,a[i]);
   tmp = swWindow.nextFrame(b,2*N);
   for(i = 0; i < N; i++)
     memcpy(output, tmp[i], 4);
 }
}
```

Figure 3-3: Ogg Vorbis C++ pseudocode.

Figure 3-1.a shows the organization of a single-threaded C++ implementation of the Vorbis CODEC. Using profiling tools, we observe that most of the execution time is spent computing an Inverse Fast Fourier Transform (IFFT) ❶. This "hot spot" is a natural candidate for FPGA acceleration.Performance bottlenecks are not the only motivation for specialized hardware, and in the cases where it does not dramatically improve throughput, it may be able to compute the result using far less energy. The cost of data-transfer both in

terms of energy and time must also be taken into consideration when selecting the accelerated kernel, since it may overshadow the improvements in the computation alone.

3.1.2 Defining the Interface and Implementing the HW

We will forgo a discussion of the mechanics of re-implementing the `ifft` function as a Verilog module, and concentrate only on the interface. The IFFT interface ❷ we choose must reflect the flexible nature of the application, which can accommodate a range of frame sizes. Shown in Figure 3-4, this interface reflects the flexible nature of the implementation, which can accommodate a range of frame sizes with dynamic flow control on both input and output data ports. To configure the module, the user must first set the frame size, which is accomplished by asserting the appropriate value on the frameSz input wires, and asserting frameSzEn. Attempting to configure the module when frameSzRdy is low results in undefined behavior. After setting the frame size, the input frame can be inserted one word at a time on the dataInput wires. A similar hand-shaking protocol involving dataInputRdy/dataInputEn is implemented which applies back-pressure and maintains correct flow-control. The implementation keeps track of the number of words that have been received, and once a full frame is present, it computes the IFFT and subsequently makes the data available on the output port. Once again, dataOutputRdy/dataOutputValid implement a hand-shaking protocol enforcing flow-control as data is streamed, one word at a time, from the IFFT block. The interface and associated timing properties are quite conventional from a hardware perspective, but making the connection from this interface to a function call in SW will require substantial effort.

Implementing the specialized HW: In general, specialized HW tends to be more efficient, in terms of energy or time, than SW implementations. This effect is especially acute for highly parallel algorithms, like IFFT, which will perform faster on an FPGA than in SW ❸.

3.1.3 Data-Type Conversion

Mismatch of data representations is a common source of errors in HW/SW codesign. The objects generated by the application must be correctly converted to the bit representation

```
module IFFT ( clock,              frameSz,
              frameSzRdy,         frameSzRdy,
              dataInput,          dataInputEn,
              dataInputRdy,       dataOutput,
              dataOutputRdy,      dataOutputEn );

  input          clock;
  input [3:0]    frameSz;
  input          frameSzEn;
  output         frameSzRdy;
  input [31:0]   dataInput;
  input          dataInputEn;
  output         dataInputRdy;
  output [31:0]  dataOutput;
  output         dataOutputRdy;
  input          dataOutputEn;
  ...
```

Figure 3-4: IFFT Verilog Interface

expected by the accelerator. This problem is compounded by the fact that the C++ compiler and the Verilog compiler may have completely different layouts for the "same" data structure:

C++:

```
template<typename F, typename I> struct FixPt{ F fract; I int; };
template<typename T> struct Complex{T rel; T img;};
```

Verilog:

```
typedef struct {bit[31:0] fract; bit[31:0] int;} FixPt;
typedef struct {FixPt rel; FixPt img;}; Complex_FixPt;
```

The two languages may have different endian conventions, and even two compilers for the same language can use different conventions depending on the targeted substrate. In addition, there are SW structures such as pointers which cannot be directly represented in HW. Maintaining consistency between two separate representations of the same data-types is both tedious and error-prone.

3.1.4 Connecting To The Communication Fabric

Tight FPGA/Microprocessor integration is generally achieved using some kind of bus, and in order to expose the full Hardware (HW) functionality to the Software (SW), all the ports must ultimately be connected to the bus in one form or another. The high-level goal is to provide a simple abstraction of the accelerated HW kernel to the SW application which hides the more complicated aspects of the HW interface protocol, such as timing. The hand-shaking protocol which we implemented on each logical port of the IFFT accelerator is very similar to many bus protocols, so we have the option of connecting the device to the bus using three separate slave interfaces: two for writing and one for reading. In spite of the similarity, we will need to implement a thin layer or shim to sit between the IFFT HW and the bus to correctly convert between the two hand-shaking protocols.

The next consideration is on the SW side, and relates to flow-control and how best to expose the HW functionality. What happens if SW attempts to read invalid data from the output port, or write data without a valid Ack signal: should these invocations block or should we expose this in some other way? Most bus protocols support blocking read/write functionality, which we will apply to the Vorbis example. The details of how to do this differ with every system; for example the Zynq platform has some libraries implemented by Xilinx which expose this functionality as C++ macros. To hide the implementation details, we will define three functions `frameSz()`, `dataOutput()`, and `dataInput()`, whose bodies implement blocking reads and writes to HW. If the application SW is run in a traditional operating system, this would require a very thin driver (in kernel space) which the application could invoke using **ioctl**. If no OS is present the application can link these routines directly.

Having arrived at a suitable SW abstraction of the FPGA accelerator, we will now integrate it into the application code. The invocation of the SW implementation of IFFT in Figure 3-3 (`ifft(2*N, a, a)`) can be replaced by the code shown in Figure 3-5.

The transformed code will now exploit the HW accelerator to compute the correct values as shown in Figure 3-1.b, but given the latency introduced by transferring the data to and from the accelerator over the bus one word at a time, it is highly unlikely that the

```
...
hwIFFT.setSize(2*N);
for(i = 0; i < 2*N; i++)
  hwIFFT.dataInput(a[i]);
for(i = 0; i < 2*N; i++)
  hwIFFT.dataOutput(&a[i]);
...
```

Figure 3-5: Naive Integration of Accelerated Kernel

overall throughput algorithm will improve by using the accelerator in this manner ❹.

3.1.5 Restructuring The Application SW

In almost all such HW designs where significant data transfer is involved, it is important to find ways to speed up the communication. It is often impossible to change the *latency* of communication because it is tied to system components like bus and networks which are typically fixed, but several potential solutions exist to hide this latency. Chief among these are increasing the granularity of communication, and introducing pipelining.

Communication Granularity: Transferring data one word at a time is inefficient, given the overhead of a bus transaction. This cost can be amortized by transferring larger blocks of data directly into the accelerator memory. On systems with dedicated DMA hardware, this change will significantly reduce transmission costs, but care must be taken to ensure that sufficient buffering exists before the burst transfer is initiated ❺. On some systems, failure to ensure this condition raises the possibility of deadlock: if the HW accelerator exerts back pressure during a burst transaction, the DMA engine will stall until the HW begins accepting tokens again. If bus control cannot be transferred (as is the case with many simpler bus implementations), the entire system may deadlock if the HW is not able to make forward progress.

In the case of our IFFT implementation, we will enable the transfer of data at the frame granularity. It is possible to ensure that the IFFT implementation implements the correct buffering guarantees, but a simpler approach which doesn't require us to modify the implementation lends itself to cases where we may have received the IP for the accelerator from

a third party. To ensure the correct buffering conditions, we will add a thin layer, or shim, on top of the accelerator interface. The shim consist of buffering and some control logic which ensures that transactions will begin only when certain conditions are met. The input buffering is augmented with control logic which begins accepting words only if it contains enough free space to accept an entire frame. Likewise, the control logic surrounding the output buffering begins streaming the transformed data only when it has accumulated an entire frame from the IFFT block. The Vorbis example is relatively simple, but for applications with more data-paths crossing the shared bus and complex control, we may also require some kind of scheduling logic to enforce fairness.

With the shims in place, we can implement the functions `putFrame()` and `getFrame()` which invoke a burst write and read respectively, and replace the two loops in Figure 3-5 which invoke `hwIFFT.dataInput()` and `hwIFFT.dataOutput()` with these new functions (see Figure 3-6).

Pipelining: Pipelining the HW accelerator hardware can further improve the throughput [28] ❻. This will require substantial transformation of the Verilog implementation, but once complete, we can exploit this concurrency in SW through the use of multi-threading, as shown in Figure 3-1.c using the code in Figure 3-6. The pipelined HW implementation can now begin receiving new frames even though it may not have completed the transformation of the previous ones. A separate thread is forked which invokes `backend_complete()` to drain the HW accelerator and drive the final audio pipeline stages.

Nonblocking Interface: Instead of implementing blocking function calls, we could also have chose non-blocking variants, which return an error code if the data is not ready, or there is insufficient buffering. SW is always free to take advantage of the error codes returned by the interface methods to do other useful work. Using a reactive programming paradigm might be a good way to improve the overall efficiency of the application ❼.

```
void vorbis(file* input, void* output){
 Parser p(input);
 int N = p.frameSize();
 fork(backend_complete(N));
 while(!p.eof()){
   Floor f(p);  Residue r(p);
   Frame frame(f,r);
   int a[2*N], i;
   for(i = 0; i < N; i++){
     a[i]   = imdctPreLo(i,N,frame[i]);
     a[i+N] = imdctPreHi(i,N,frame[i]);}
   hwIFFT.setSize(2*N);
   hwIFFT.putFrame(a,2*N);
 }
}
void backend_complete(int N){
 while(true){
  int a*, b[2*N], i, *tmp;
  hwIFFT.getFrame(&a);
  for(i = 0; i < N; i++)
    b[bitReverse(i)] = imdctPost(i,N,a[i]);
  tmp = swWindow.nextFrame(b,2*N);
  for(i = 0; i < N; i++)
   memcpy(AUDIO_DEV, tmp[i], 4);
 }
}
```

Figure 3-6: Multi-threaded use of pipelined HW

3.2 An Incremental Methodology Using Dataflow

In this section, we introduce a framework for HW-SW communication. We examine the possibility of automating the steps detailed in Section 3.1 and to what extent our proposed programming model improves matters.

Our methodology requires the high-level organization of the algorithm to be specified in terms of a dataflow graph [42]. In the dataflow model, nodes in the graph encapsulate computation, while directed edges represent communication. In order to execute the functionality at a node, data must be present on the input edges and buffering must be available on the output edges. The edges themselves are implemented as *guarded* FIFO channels,

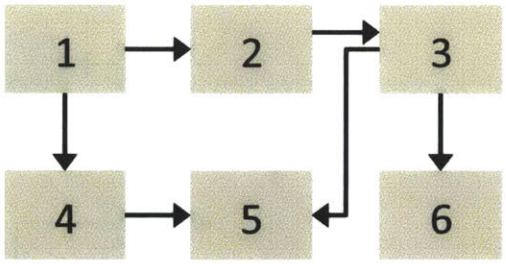


Figure 3-7: Structure Application as Dataflow Graph

which means that they respect the dataflow contract and naturally enforce flow control. Figure 3-7 shows an application which has been decomposed into a dataflow graph consisting of six nodes. The edges in the graph reflect the underlying communication between application components. Once the application has been reorganized, the programmer must profile its execution and select kernels for HW acceleration, as shown in Figure 3-8. The coloring indicates the partitioning choice; nodes colored blue have been selected for HW implementation. Rather than interlinked execution with a blocking RPC, interaction with the HW accelerator now involves enqueueing data into output FIFOs and dequeuing data from input FIFOs.

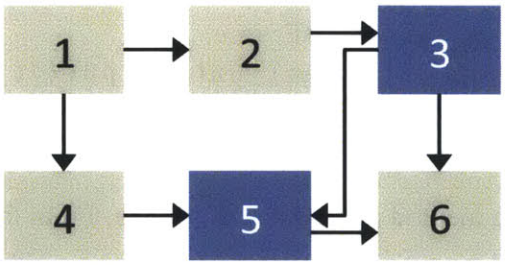


Figure 3-8: Profile in SW and Select Kernels for HW Acceleration

It is important to note that the dataflow edges effectively isolate the functionality encapsulated by the nodes. When migrating functionality from SW to HW, it may need to be reimplemented in a different language, e.g. from C++ to Verilog, but as long as the functionality is selected at the same granularity as the nodes in the graph, this migration will not affect adjacent nodes. This isolation allows the designer to lower the reimplementation cost when experimenting with different HW/SW partitions.

As shown in Figure 3-9, the HW/SW interface is simply the union of all dataflow edges

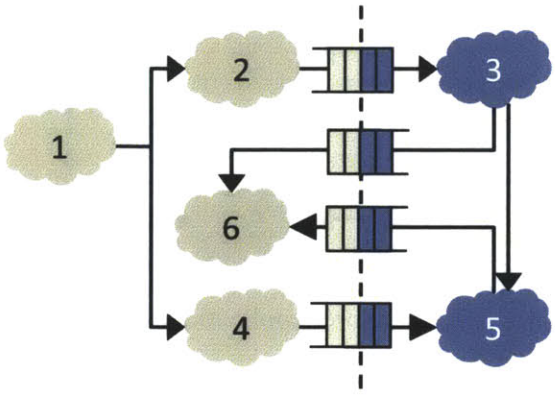


Figure 3-9: Accumulate Dataflow Edges to form HW-SW Interface

which cross the HW/SW boundary. We use Bluespec [10] to specify the node interfaces, since its guarded interface semantics provide a direct implementation of the dataflow contract. Below we give the Bluespec interface definition for the IFFT module discussed in Section 3.1:

```
typedef FixedPoint#(8,24) FxPt;
typedef Vector#(MAX_IFFT_SZ, FxPt) Frame;

interface IFFT
  method Action dataInput(Frame x);
  method ActionValue#(Frame) dataOutput();
  method Action frameSz(Bit#(3) x);
endinterface
```

We assign one interface method to each dataflow edge. From this declaration, an interface compiler could easily infer the FIFO channels connecting HW and SW, the conversion routines for all data-types being transmitted, and the correct abstraction to expose to the SW application code.

Though we use Bluespec to define the communication interfaces, the user can choose the implementation language at each node. For example, it is possible to implement the body of the HW accelerator in Verilog or VHDL, but care must be taken to ensure correct enforcement of the guarded interface semantics. On the other hand, using Bluespec to specify the entire module guarantees the guarded interface will be correctly implemented by construction.

Figure 3-9 shows the dataflow graph in a state where the nodes have been grouped to form the HW and SW application components. The task of connecting these components is simply a matter of implementing the FIFO channels on the shared communication fabric, and presenting a usable abstraction to the application-level HW and SW. Due to the latency tolerance of dataflow edges, we have some flexibility which can be exploited in the interest of efficiency.

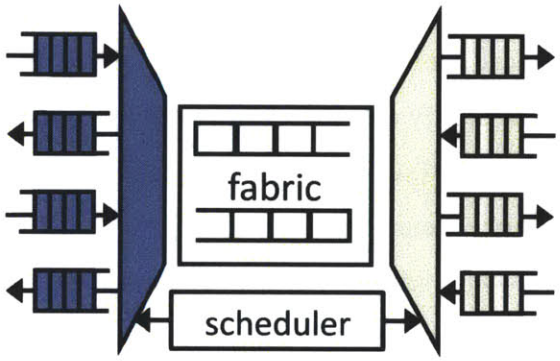


Figure 3-10: Synthesize Interface Implementation

The first step in implementing the FIFOs used in the programming model is to map them to the shared communication fabric. The output of the synthesis process is illustrated in Figure 3-10. Given a model of the fabric, marshaling and demarshaling routines are generated for each data-type being transmitted. A scheduler is also generated to ensure that all virtual FIFOs are fairly serviced. The scheduling logic is especially important when multiple CPU cores are concurrently accessing different interface FIFOs over the same bus.

So as not to introduce deadlocks, per-FIFO buffering is added to both HW and SW domains. The buffering at the source and destination of each virtual FIFO must be sufficient to store a complete high-level message. If a channel is using bursty communication, this message consists of a complete burst, otherwise it is a single instance of the transmitted type. The scheduler tracks the states of both the source and destination buffers and transmissions are scheduled when sufficient data has been accumulated in the source buffer and sufficient space exists in the destination. Because the scheduler keeps track of both source and destination buffer states, we can guarantee that this approach will not introduce

deadlocks.

To simplify the logic, we assume a static priority among the channels, though it is easy to imagine a more complex scheduler which reacts to the environment dynamically. If the user does not specify otherwise, we treat all FIFOs with the same priority and implement a simple round-robin schedule. Once an abstract model of the physical network has been manually specified, this synthesis problem can be automated completely.

The final step, shown in Figure 3-11, requires the implementation of the synthesized components on a physical platform. As with the interface definition and infrastructure synthesis, this too can be totally automated, provided the HW interface abstractions used by the synthesized code are present. To promote platform portability, this can be further abstracted into something we refer to as *Platform Support*. Platform support is simply a set of low-level RTL and C++ libraries which provide a standardized interface to the physical components of the chip on which we will execute the application.

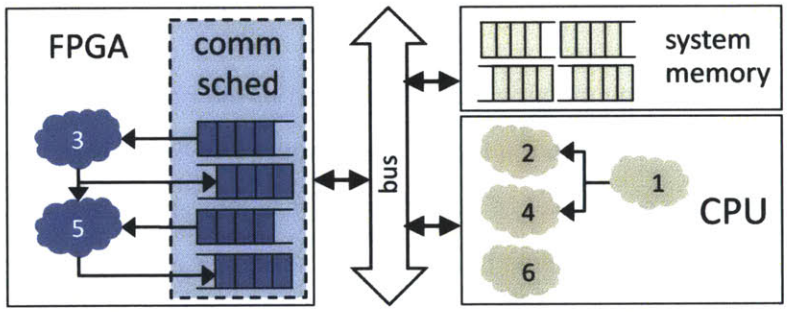


Figure 3-11: Map Synthesized Infrastructure to Physical Platform

3.3 Interface Compiler

From our discussion of the methodology, it follows naturally that once an application has been organized into a dataflow graph and nodes have been designated as either SW or HW, an interface compiler can automatically generate the low-level SW driver code and HW shims along the lines of our discussion in Section 3.1. There is a lot of interesting work which can be done to improve the quality of the generated interfaces, some of which was demonstrated in the context of connecting two HW partitions executed on different

FPGAs [36]. In this thesis, we take a very pragmatic approach and push the infrastructure only as far as necessary to generate results of sufficient quality.

The primary challenge in efficient communication is the accumulation of sufficiently sized bursts to amortize the cost of the underlying bus transaction. Due to the dynamic nature of the dataflow model, inferring these automatically is problematic. Our first approach, which proved quite practical, was to give the programmer the ability to specify a burst size for each interface FIFO and to synthesize the infrastructure accordingly. The language-level interface FIFOs must be used in a latency-insensitive manner. This is a property which we do not check in the compiler; we rely on the programmer to ensure there is no implicit reliance on a particular latency which could cause deadlock in the implementation. In the same vein, we rely on the programmer to ensure that implementing the specified burst-size will not introduce deadlocks either.

If the SW is generated using BCL, the programmer can also enable a dynamic burst accumulator if he is unsure about his choice of burst size. If this is turned on, the run-time will attempt to accumulate a full burst within a particular time slice. If a full burst is not accumulated, the partial burst will be transmitted in order to avoid head-of-line blocking. This dynamic approach achieved reasonable performance, but proved less useful since burst sizes for the high-traffic edges are often obvious (frame granularity, for example). Low-traffic edges are not performance critical, so a default burst-size of one will not negatively impact application performance.

After determining the burst size of each channel, the next task is to map the dataflow FIFOs to the physical communication fabric. We use an abstract model of the fabric which consists of m distinct channels, each of which has a known bandwidth and latency. Due to the dynamic nature of the programs we are compiling, we can make no assumptions about the bandwidth or latency requirements of the n dataflow FIFOs we are attempting to map to the physical channels. Because of this, an optimal mapping of the n FIFOs to the m physical channels is impossible, and we rely on a heuristic based approach.

The interface compiler maps the dataflow FIFOs to the channels available in the physical platform by applying some simple heuristics. These heuristics are effective primarily due to the relative simplicity of the communication fabric: on the Zedboard we used a sin-

gle AXI-Streaming bus to connect the HW and SW components, while on the ML507 we used the four independent HDMA channels. Our greedy heuristic sorts the FIFO edges by the relative weights of their communication requirements. Since this is a dynamic property we use burst-size as an approximation for traffic. Beginning with the heaviest FIFOs, we assign them to physical channels in a round-robin manner in an attempt to distribute the bandwidth evenly. This course heuristic will likely fail as the communication fabrics become more complex, but it proved effective for the platforms and applications we considered.

Finally, we need to synthesize the infrastructure to implement the burst sizes and lane assignments. We generate ring-buffers for each virtual channel in SW, and assign BRAM buffering in HW. A communication scheduler is implemented in the FPGA fabric to arbitrate the use of each shared physical channel. The scheduler is notified by the application components when data is ready to be sent (in either direction), and these transactions are initiated according to a static priority. The default is to assign equal priority to all channels, though the programmer can override this with his own static priority. The overhead of this generated infrastructure in terms of FPGA usage was negligible in each of the benchmarks we considered.

3.4 Using A Unified Language

Ultimately, a programmer can choose to use a unified language to specify the dataflow and computation of the entire program, and rely on a compiler to automatically generate both the HW and SW. If this approach is taken, the reimplementation effort will be minimal when migrating the modules from SW to HW, or vice versa. Additionally, the reorganization of the SW to accommodate the movement of functionality to HW is automatic. This is the subject of Chapters 4 and 5.

Chapter 4

HW/SW Codesign in BCL

In this chapter we will begin by introducing the Bluespec Codesign Language (BCL) after which we will demonstrate its use through the Vorbis pipeline example and show how a BCL design is partitioned between hardware and software. We end by showing how a BCL design can be integrated into the infrastructure presented in Chapter 3.

4.1 A Language of Guarded Atomic Actions

Based on the hardware description language Bluespec System Verilog, BCL is a modern statically-typed language with the flexibility to target either hardware or software. It contains higher-order functions and the rich data structures required to express fine-grain parallelism. Any language with hardware as a target must be restricted so that it is compilable into efficient FSMs. Consequently BCL can be used for writing only the type of software which does not require dynamic heap storage allocation and where the stack depth is known at compile-time. In fact, in a BCL program all state must be declared *explicitly*, and the type system prohibits dynamic allocation of objects. Because BCL's intended use is for describing software primarily at the bottom of the stack, this restriction is not burdensome.

In BCL, behavior is described using guarded atomic actions (GAAs) or *rules* [39]. Each rule specifies a state transition (its *body*) on the state of the system and a predicate (a *guard*) that must be valid before this rule can be executed, *i.e.,* the state transformation can take place. One executes a program by randomly selecting a rule whose predicate is

program ::=	**Program** *name* [*m*] [**Rule** *R*: *a*]	// A list of Modules and Rules
m ::=	[**Register** *r* (v_0)]	// Reg with initial values
||	**Module** *name*	
	[*m*]	// Submodules
	[**ActMeth** $g = \lambda x.a$]	//Action method
	[**ValMeth** $f = \lambda x.e$]	//Value method
v ::=	*c*	// Constant Value
||	*t*	// Variable Reference
a ::=	*r* **:=** *e*	// Register update
||	**if** *e* **then** *a*	// Conditional action
||	*a* | *a*	// Parallel composition
||	*a* **;** *a*	// Sequential composition
||	*a* **when** *e*	// Guarded action
||	(*t* = *e* **in** *a*)	// Let action
||	**loop** *e a*	// Loop action
||	**localGuard** *a*	// Localized guard
||	$m.g(e)$	// Action method call g of m
e ::=	*r*	// Register Read
||	*c*	// Constant Value
||	*t*	// Variable Reference
||	*e op e*	// Primitive Operation
||	*e* **?** *e* **:** *e*	// Conditional Expression
||	*e* **when** *e*	// Guarded Expression
||	(*t* = *e* **in** *e*)	// Let Expression
||	$m.f(e)$	// Value method call f of m
op ::=	&& | || | ...	// Primitive operations

Figure 4-1: Grammar of BCL. For simplicity we will assume all module and method names are unique.

valid and executing its body. Any possible sequencing of rules is valid; the implementation is responsible for determining which rules are selected and executed.

The reader will recognize the similarities between BCL and transactional memory systems, though there are some significant semantic differences. Both rely on atomicity (of the transactions or actions), which provides natural semantics for such systems since the

behavior of a parallel program can always be understood in terms of some sequential execution of atomic actions. Using the jargon of transactional memories, a transaction either succeeds (commits all its variable updates) or fails (behaves like a "no-op"). The idea of "optimistic concurrency" is also very important in Transactional Memory (TM) systems. Optimistic concurrency describes a situation where many transactions try to execute simultaneously, though some may have to retry when a conflict with another atomic transaction is detected. The notions of concurrency and atomicity in TM systems apply to BCL programs as well. BCL guarded atomic actions differ from transactional memories in several important ways:

1. BCL atomic actions have explicit internal parallelism, meaning that the language used to describe the atomic actions is not a sequential language. This is extremely important since we are based on an HDL which naturally exploits highly parallel sub-actions. Similarly, HDLs are built on the idea that users specify all the updates that are made at the end of each clock cycle, which corresponds to the behavior of BCL.

2. BCL atomic actions are influenced by the idea that in synchronous hardware systems, registers are read at the beginning of a clock cycle and updated at the end of the clock cycle. This means that the actions that can be performed in one clock cycle do not require any explicit shadow state. For example, one can perform the swap of two registers in one atomic action without needing any temporary variable.

3. Guards in atomic actions indicate when expression or actions are invalid. This provides a mechanism for safely composing parallel atomic actions. The safe use of methods in a guarded atomic actions is enforced through the use of modules with guarded interfaces. Guards can cause an action can fail even when it actually is executing in isolation.

These properties of guarded atomic actions create new challenges in software compilation and form the core of BCL, whose grammar is shown in Figure 4-1. This language is produced after all the meta-programming features have been evaluated.

A BCL program consists of a name, a set of a modules, and a set of rules. Each BCL module consists of a set of (sub)modules and sets of *action methods* and *value methods* which are called by methods of the enclosing module or rules in the top-level program. **Register** is a primitive module with special syntax for calling its read and write methods. Though rules are allowed at all levels of the module hierarchy in BCL, we restrict the grammar to allow rules only at the top level. This restriction is only to simplify the discussion of the language semantics and does not come at the cost of generality, since all rules can be lifted to the top programatically by repeatedly replacing a rule in a submodule with an action method containing the body of the rule and a rule that calls just that action method.

We refer to the collection of all registers in a BCL program as its *state*. The evaluation of a BCL rule produces a new value for the state and a boolean guard value which specifies if the state change is permitted. Every rule in BCL is deterministic in the sense that the guard value and state change computed when evaluating a rule are a function of the current state. The execution of a BCL program can be described as follows:

Repeatedly:

1. Choose a rule to execute.
2. Compute the set of state updates and the value of the rule's guard.
3. If the guard is true, apply the updates.

Figure 4-2: Execution Semantics of Bluespec

Since this procedure involves a nondeterministic choice and the choice potentially affects the observed behaviors, our BCL program is more like a specification as opposed to an implementation. To obtain an effective implementation we selectively restrict the model to limit nondeterminism and introduce a notion of fairness in rule selection.

4.1.1 Semantics of Rule Execution

For the sake of completeness, we include a discussion of the semantics of BCL. These were originally presented in the context of hardware description [25], and an extensive

discussion also appears in appears in [29]. The operational semantics of a rule execution in BCL are given using SOS-style evaluation rules in Figures 4-3, 4-4, and 4-5. The meaning of each composite atomic action will be explained in terms of its constituent actions.

Each rule is evaluated in an environment consisting of the triplet $\langle S, U, B \rangle$, where S represents the values of all the state *before* the rule execution, U is a list of key/value pairs representing updates which will be applied to S upon successful completion of the rule (an empty set implies no updates will be applied), and B contains the locally-bound variables in the scope of the action or expression being evaluated. When evaluating a rule, U and B are initially empty, and S contains the values of all the register state. Conflicting updates to the same register produce a *dynamic error*, shown in the figures as "DUE", and have the same effect as a guard failure on the application of a rule's updates. The semantic rules progressively build the effects of a rule during its evaluation by composing the effects of its constituent actions. For correct composition, the evaluation of a guarded expression which is not ready can return the value "NR", which can be stored in a binding but *cannot* be assigned to a register.

The semantic machine is incomplete in the sense that there are cases where the execution gets *stuck* because none of the rules in Figure 4-3 apply. In such cases we will say that the action produced no updates. This allows us to present a much more succinct set of rules which are not cluttered by having to deal with $\perp$ propagation.

Action Composition

Figure 4-3 gives two ways to compose actions together: *parallel composition* and *sequential composition*. If two actions $A_1 | A_2$ are composed in parallel both observe the same initial state and do not observe each other's updates. Thus the action $r_1 := r_2 \mid r_2 := r_1$ swaps the values in registers r_1 and r_2. Since rules themselves are deterministic, there is never any ambiguity due to the order in which sub-actions complete. If two actions update the same state element then they cannot be composed in parallel. Because of conditional actions one can only determine approximately if a parallel composition is legal. However, it is preferable if such an error is disallowed by static checking in an earlier compilation step.

reg-update
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow v,\ v \neq \text{NR}}{\langle S,U,B\rangle \vdash r := e \Rightarrow \{r \mapsto v\}}$$

if-true
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \mathit{true},\ \langle S,U,B\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash \textbf{if } e \textbf{ then } a \Rightarrow U'}$$

if-false
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \mathit{false}}{\langle S,U,B\rangle \vdash \textbf{if } e \textbf{ then } a \Rightarrow \{\}}$$

a-when-true
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \mathit{true},\ \langle S,U,B\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash a \textbf{ when } e \Rightarrow U'}$$

par
$$\frac{\langle S,U,B\rangle \vdash a_1 \Rightarrow U_1, \langle S,U,B\rangle \vdash a_2 \Rightarrow U_2,\ U_1 \neq \text{NR}, U_2 \neq \text{NR}}{\langle S,U,B\rangle \vdash a_1 \mid a_2 \Rightarrow U_1 \uplus U_2}$$

seq-DUE
$$\frac{\langle S,U,B\rangle \vdash a_1 \Rightarrow \text{DUE},}{\langle S,U,B\rangle \vdash a_1 \textbf{ ; } a_2 \Rightarrow \text{DUE}}$$

seq
$$\frac{\langle S,U,B\rangle \vdash a_1 \Rightarrow U_1, U_1 \neq \text{NR}, U_1 \neq \text{DUE}\quad \langle S,U{+\!+}U_1,B\rangle \vdash a_2 \Rightarrow U_2, U_2 \neq \text{NR}}{\langle S,U,B\rangle \vdash a_1 \textbf{ ; } a_2 \Rightarrow U_1{+\!+}U_2}$$

a-let-sub
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow v, \langle S,U,B[v/t]\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash t = e \textbf{ in } a \Rightarrow U'}$$

a-meth-call
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow v,\ ,v \neq \text{NR},\quad m.g = \langle \lambda t.a\rangle,\ \langle S,U,B[v/t]\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash m.g(e) \Rightarrow U'}$$

a-loop-false
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \mathit{false}}{\langle S,U,B\rangle \vdash \textbf{loop } e\ a \Rightarrow \{\}}$$

a-loop-true
$$\frac{\langle S,U,B\rangle \vdash e \Rightarrow \mathit{true}\ , \langle S,U,B\rangle \vdash a \textbf{ ; loop } e\ a \Rightarrow U'}{\langle S,U,B\rangle \vdash \textbf{loop } e\ a \Rightarrow U'}$$

a-localGuard-fail
$$\frac{\langle S,U,B\rangle \vdash a \Rightarrow \text{NR}}{\langle S,U,B\rangle \vdash \textbf{localGuard } a \Rightarrow \{\}}$$

a-localGuard-true
$$\frac{\langle S,U,B\rangle \vdash a \Rightarrow U', U' \neq \text{NR}}{\langle S,U,B\rangle \vdash \textbf{localGuard } a \Rightarrow U'}$$

Figure 4-3: Operational semantics of a BCL Actions. When no rule applies the action evaluates to NR. Rule bodies which evaluate to NR produce no state update.

Sequential composition is more in line with other languages with atomic actions. The action $A_1; A_2$ represents the execution of A_1 followed by A_2. A_2 observes the full effect of A_1. No other action observes A_1's updates without also observing A_2's updates.

reg-read	$\langle S, U, B\rangle \vdash r \twoheadrightarrow (S{+}{+}U)(r)$
const	$\langle S, U, B\rangle \vdash c \Rightarrow \underline{c}$
variable	$\langle S, U, B\rangle \vdash t \Rightarrow B(t)$
op	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow v_1,\ v_1 \neq \textbf{NR} \quad \langle S, U, B\rangle \vdash e_2 \Rightarrow v_2,\ v_2 \neq \textbf{NR}}{\langle S, U, B\rangle \vdash e_1\ op\ e_2 \Rightarrow v_1\ \underline{op}\ v_2}$
tri-true	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow true,\ \langle S, U, B\rangle \vdash e_2 \Rightarrow v}{\langle S, U, B\rangle \vdash e_1\ \textbf{?}\ e_2\ \textbf{:}\ e_3 \Rightarrow v}$
tri-false	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow false,\ \langle S, U, B\rangle \vdash e_3 \Rightarrow v}{\langle S, U, B\rangle \vdash e_1\ \textbf{?}\ e_2\ \textbf{:}\ e_3 \Rightarrow v}$
e-when-true	$\dfrac{\langle S, U, B\rangle \vdash e_2 \Rightarrow true,\ \langle S, U, B\rangle \vdash e_1 \Rightarrow v}{\langle S, U, B\rangle \vdash e_1\ \textbf{when}\ e_2 \Rightarrow v}$
e-when-false	$\dfrac{\langle S, U, B\rangle \vdash e_2 \Rightarrow false}{\langle S, U, B\rangle \vdash e_1\ \textbf{when}\ e_2 \Rightarrow \textbf{NR}}$
e-let-sub	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow v_1, \langle S, U, B[v_1/t]\rangle \vdash e_2 \Rightarrow v_2}{\langle S, U, B\rangle \vdash t = e_1\ \textbf{in}\ e_2 \Rightarrow v_2}$
e-meth-call	$\dfrac{\langle S, U, B\rangle \vdash e \Rightarrow v,\ v \neq \textbf{NR}, \quad m.f = \langle \lambda t.e_b\rangle,\ \langle S, U, B[v/t]\rangle \vdash e_b \Rightarrow v'}{\langle S, U, B\rangle \vdash m.f(e) \Rightarrow v'}$

Figure 4-4: Operational semantics of a BCL Expressions. When no rule applies the expression evaluates to NR

Merge Functions:

$$
\begin{array}{rl}
L_1{+}{+}(\text{DUE}) & = \text{DUE} \\
L_1{+}{+}(L_2[v/t]) & = (L_1{+}{+}L_2)[v/t] \\
L_1{+}{+}\{\} & = L_1 \\
U_1 \uplus U_2 & = \text{DUE if } U_1 = \text{DUE or } U_2 = \text{DUE} \\
 & = \text{DUE if } \exists r.\{r \mapsto v_1\} \in U_1 \wedge \{r \mapsto v_2\} \in U_2 \\
 & \quad \text{otherwise } U_1 \cup U_2 \\
\{\}(x) & = \bot \\
S[v/t](x) & = v \text{ if } t = x \text{ otherwise } S(x)
\end{array}
$$

Figure 4-5: Helper Functions for Operational Semantics

Conditional versus Guarded Actions

BCL has both conditional actions ("if") as well as guarded actions ("when"). These are similar as they both restrict the evaluation of an action based on some condition. The difference is their scope of effect: conditional actions have only a local effect whereas

guarded actions have a global effect. If the predicate of an "if" evaluates to false, then that action doesn't happen (produces no updates). If a "when" predicate is false, the sub-action (and as a result the whole atomic action that contains it) is invalid. One of the best ways to understand the differences between "when" and "if" is to examine the axioms in Figure 4-6.

Axioms A.1 and A.2 collectively state that a guard on one action in a parallel composition affects all the other actions. Axiom A.3 deals with a particular sequential composition. Axioms A.4 and A.5 state that guards in conditional actions are reflected only when the condition is true, but guards in the predicate of a condition are always evaluated. A.6 deals with merging "when" clauses. A.7 and A.8 translate expression "when"-clauses to action "when"-clauses. Axiom A.9 states that top-level "when" in a rule can be treated as an "if" and vice versa.

A.1	$(a_1$ **when** $p) \mid a_2$	$\equiv$	$(a_1 \mid a_2)$ **when** p
A.2	$a_1 \mid (a_2$ **when** $p)$	$\equiv$	$(a_1 \mid a_2)$ **when** p
A.3	$(a_1$ **when** $p)$; a_2	$\equiv$	$(a_1$; $a_2)$ **when** p
A.4	if $(e$ **when** $p)$ **then** a	$\equiv$	(if e then a) **when** p
A.5	if e then $(a$ **when** $p)$	$\equiv$	(if e then a) **when** $(p \vee \neg e)$
A.6	$(a$ **when** $p)$ **when** q	$\equiv$	a **when** $(p \wedge q)$
A.7	$r := (e$ **when** $p)$	$\equiv$	$(r := e)$ **when** p
A.8	$m.h(e$ **when** $p)$	$\equiv$	$m.h(e)$ **when** p
A.9	Rule n if p then a	$\equiv$	Rule n $(a$ **when** $p)$

Figure 4-6: When-Related Axioms for Actions

Strict Method Calls and Non-Strict Lets

We have chosen non-strict lets and function calls because they permit more useful algebraic laws for program transformation. However, we have chosen strict method calls because each method represents a concrete resource in our implementation. Additionally, modular compilation would be impossible (or greatly complicated) without this exception. With lazy method calls, all interface types would need to be augmented with an implicit valid bit, while the propagation of failure ($\perp$) would be ill-defined, especially in the case of a

heterogeneous (HW/SW) implementation. These facts can be seen in our SOS rules in Figure 4-3, and 4-4. Notice that both a-let-sub and e-let-sub rules store the value of an expression, even if it was NR, *i.e.*,, $\perp$, in the bindings B. However, such a value cannot be stored in a register (see reg-update rule), passed to an expression (see a-meth-call and e-meth-call rules) or used by a primitive operation (see op rule).

4.2 The Vorbis Pipeline in BCL

Having given a formal description of the BCL Language in the previous section, we will now demonstrate it's use on a concrete problem. In Figure 4-7 we give simplified version of the code implementing the back-end of the Vorbis pipeline introduced in Chapter 2.

```
module mkVorbisBackEnd(VorbisBackEnd#(k,t))
  IFFT#(2*k, Complex#(t)) ifft <- mkIFFT;
  Window#(2*k, t)         window <- mkWindow;

  method Action input(Vector#(k,Complex#(t)) vx)
    Vector#(2*k,Complex#(t)) v;
    for(int i = 0; i < k; i++)
      v[i]   = preTable1[i]*vx[i];
      v[K+i] = preTable2[i]*vx[i];
    ifft.input(v);

  rule xfer
    let x = ifft.output() in (ifft.deq();
    Vector#(2*k,t) v;
    for(int i = 0; i < 2*k; i++)
      v[i] = x[bitReverse(i)].real;
    window.input(v))

  rule output
    let rv = window.output();
    window.deq(); AUDIO_DEV.output(rv)
endmodule
```

Figure 4-7: BCL Specification of the Vorbis Back-End

A BCL program consists of a hierarchy of modules, and as with any object-oriented

language, all interactions with modules occur through their interface methods. The mkVorbisBackend module definition instantiates two sub-modules: "ifft" and "window". Each module is potentially stateful, but ultimately all state is built up from primitive elements called *registers*. It is important not to confuse the state elements with ordinary variables such as "x" or "v", which are just names. Like functional languages, "x = exp" simply assigns the name "x" to the expression "exp".

Module interfaces are declared separately from the implementation to encourage reuse; the following code defines the polymorphic IFFT interface, parameterized by the type variable 't', where 't' is used to indicate the type of components (*e.g.*,'Fix32', 'Fix16', 'Float63') used to construct the complex numbers:

```
interface IFFT#(numeric type k, type t)
  method Action input(Vector#(k,Complex#(t)) x)
  method Vector#(k,Complex#(t))) output()
  method Action deq()
endinterface
```

In BCL, state change is indicated by the Action type. The method named "input" is of type Action and takes as arguments an input frame of size k. This was declared under the assumption that any *implementation* of this interface will store the frame in some internal buffering, changing its state and necessitating the Action type. Methods such as "output" which are not of type Action may only read state and compute the return value with a pure function.

The VorbisBackend interface, whose implementation is given in Figure 4-7, contains a single action method, input. This interface has no output because the final effect of inputting a frame is the transmission of PCM packets to the audio device, but could be easily modified to resemble the IFFT interface declared previously. In contrast to other object-oriented languages, in BCL every method has an associated *guard*, which indicates whether that method is ready to be called. An action method whose guard is false will have no effect, and a value method whose guard is false produces values which cannot be used in further computation; any computation which calls an unready method is itself not valid to be executed. The generation of control logic in the HW implementations is directly based

on guard composability.

In our example, the input method is only ready to be called once the ifft module is ready to accept the next frame, that is when the internal guard of "ifft.in" is true. We refer to this style of guard as *implicit*, though it is also possible to introduce guards *explicitly* with the **when** keyword. There is no semantic distinction between implicit and explicit guards. Since we cannot determine from outside a module when a particular method will be ready, guarded methods provide a natural abstraction for refining the timing of internal modules.

In addition to the interface methods, the state of a module can also be updated by its rules, which are all of type Action. The module mkVorbisBackEnd contains two such rules, one to transfer data between the ifft and window modules, and another to transmit PCM packets to the audio device. Just as with action methods, every rule produces a side-effect (state change) which is applied only when the rule's guard evaluates to true. The major difference between rules and Action methods are that rules are top-level objects the set of which specify the "behavior" of a program, whereas methods can only be invoked from within a rule. The rule's guard is the conjunction of all the explicit and implicit guards contained within the constituent actions and expressions. Rules also provide a natural way to express concurrency. For example the rules xfer and output may both be executable simultaneously, that is their guards both evaluate to true. Ultimately, all observed behaviors of the rule-based system must be understandable in terms of the execution procedure given in Figure 4-2.

Every rule modifies the state deterministically; non-determinism is introduced through the choice of which rule to fire. This makes a BCL program more like a design specification, and how we resolve this nondeterminism is very important for the quality of the implementation. For example, in hardware we often execute many enabled rules concurrently as long as one-rule-at-a-time semantics are not violated. In sequential SW, on the other hand, we want to compose long sequences of rules in order to exploit data locality. BCL also has the power to let the user specify any level of scheduling constraints [25], but experience shows that designers tend to leave all scheduling decisions to the compiler, except in extremely performance-critical modules.

4.3 Computational Domains

Computational domains, a part of the type system, is the mechanism by which we partition BCL designs between hardware and software. Domains are enforced by annotating every method with a domain name; each rule (or method) can refer to methods in only *one* domain. If a rule refers to methods from domain D, we can say that the rule *belongs* to domain D. For example, if a rule reads register 'a' and writes register 'b', the 'read' method of 'a' and 'write' method of 'b' must be in the same domain for the program to be a legal BCL program. A simple type invariant can be checked determine if the domain annotations are consistent.

This one-domain-per-rule restriction would seem to preclude inter-domain communication since all data flow occurs through rules. To enable inter-domain communication, primitive modules called *synchronizers*, which have methods in more than one domain, are provided. Inter-domain communication is possible only through synchronizers, thereby ensuring the absence of inadvertent (unsafe) inter-domain communication. As we show below, inserting synchronizers *is* the mechanism we provide to specify a program partition. The correct use of synchronizers allows the compiler to automatically infer the domain types on all rules and methods and safely partition a design, while an incorrect use will fail to type check, causing a compilation error.

Suppose we want to implement the IFFT in hardware and the remainder of the Vorbis back-end in software. The input and output methods of IFFT must be in the hardware domain (HW) but all other methods must be in the software domain (SW). This can be specified by introducing two synchronizing FIFOs with the following interfaces:

```
interface SyncHtoS#(type t)
 (HW) Action enq(t val)
 (SW) Action deq()
 (SW) t first()
endinterface
```

```
interface SyncStoH#(type t)
 (SW) Action enq(t val)
 (HW) Action deq()
 (HW) t first()
endinterface
```

These synchronizing FIFOs are inserted in the original Vorbis code, and two new rules are introduced in the HW domain to transmit data to and from the IFFT as shown in Figure 4-8

```
module mkPartitionedVorbisBackEnd(VorbisBackEnd#(t))
 IFFT#(Complex#(t)) ifft <- mkIFFT();
 Window#(t)         window <- mkWindow();

 SyncS2H#(Vector#(2*MAX, Complex#(t))) inSync<-mkSyncS2H;
 SyncH2S#(Vector#(2*MAX, Complex#(t))) outSync<-mkSyncH2S;

 method Action input(vx)
   Vector#(2*K,Complex#(t)) v;
   for(int i = 0; i < K; i++)
     v[i]   = preTable1[i]*vx[i];
     v[K+i] = preTable2[i]*vx[i];
   inSync.in(2*K,v)

 rule feedIFFT
   let rv = inSync.first(); inSync.deq();
   ifft.input(rv);

 rule drainIFFT
   let rv = ifft.output(); ifft.deq();
   outSync.enq(rv);

 rule xfer
   let x = outSync.first(); outSync.deq();
   Vector#(2*K,t) v;
   for(int i = 0; i < 2*K; i++)
     v[i] = x[bitReverse(i)].real;
   window.input(v);

 rule output
   let rv = window.output(); window.deq();
   AUDIO_DEV.output(rv);
endmodule
```

Figure 4-8: BCL Specification of the Partitioned Vorbis Back-End

Though moving the design from a single domain to multiple domains required modifying only a few lines to code, we have introduced buffering along the HW/SW cut. In general such buffering can change the behavior of the design. However if we restrict such changes to interfaces that are *latency-tolerant*, then such changes are correct by construction. This property of latency-tolerance enables modular refinement of a design; moving a module

from hardware to software is an example of modular refinement [27]. An algorithm to automatically verify these types of refinements is discussed in [26].

Domain Polymorphism: BCL also allows type parameters to be used in place of concrete domain names. This permits us to write very general partitioned codes where a domain can be moved from software to hardware or vise versa without changing the source code. We illustrate this next.

```
interface Sync#(type t, domain a, domain b)
 (a) Action enq(t val)
 (b) t first()
 (b) Action deq()
endinterface
```

We can then declare the internal synchronizers in the following manner, where **a** is a free type variable:

```
Sync#(Vector#(2*MAX, Complex#(t)), a, HW)
          inSync <- mkSync();

Sync#(Vector#(2*MAX, Complex#(t)), HW, a)
          outSync <- mkSync();
```

The resulting Vorbis module is fully polymorphic in its domain type. If the parameter **a** is instantiated to HW, the compiler will replace the synchronizers with lightweight FIFOs since no actual synchronization is necessary. If the domain type variable **a** is instantiated as "SW" then the synchronizers would become mkSyncHtoS and mkSyncStoH respectively. In other words, a very general partitioned code may insert more synchronizers than necessary for a specific partition, but these can be optimized by the compiler in a straightforward manner.

4.4 Generating Partitions

We can extract the code for a particular domain D by removing all the *rules* not annotated with D from the partitioned code once type checking has completed. As shown in Figure 4-9, the hardware partition of the mkPartitionedVorbisBackEnd module will contain

the rules "feedIFFT", and "drainIFFT", and the software partition will contain the remainder. Once separated, *each partition can now be treated as an distinct BCL program, which communicates with other partitions using synchronizer primitives.* The overall semantics of the original (unpartitioned) program will preserved, since the synchronizers enforce the property of *latency-insensitivity*. We refer to the dataflow graphs describing such partitioned programs as Latency-Insensitive Bounded Dataflow Networks or LIBDNs [64]. If used correctly, the Sync FIFOS in the BCL programs are equivalent to LIBDN FIFOs.

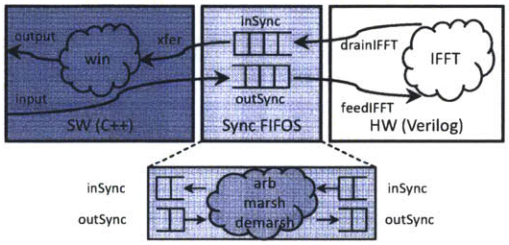


Figure 4-9: Each BCL program compiles into three partitions: HW,SW,and Interface. Each synchronizer is "split" between hardware and software, and arbitration, marshaling, and demarshaling logic is generated to connect the two over the physical channel.

4.5 Mapping Synchronizers

In addition to the generation of the HW and SW partitions as shown in Figure 4-9, the compiler must generate code to connect the two on a particular platform. A partitioned code often requires more synchronizers than the number of physical channels available in a system (typically *one*, in the form of a bus). In addition to arbitration of the physical communication substrate among multiple LIBDN FIFOs, the compiler handles the problem of marshaling and demarshaling messages. For example, in our program we specified atomic transfers at the audio-frame granularity, but the transmission of these "messages" occurs at a granularity which is dictated by the physical substrate. Through analysis, the compiler is sometimes able to transform the marshaling and demarshaling loops into more efficient burst transfers.

The interface compiler described in Chapter 3 generates the necessary software and hardware infrastructures, assuming that the HW partition fits on a single FPGA. Fleming

et al. have synthesized a a similar communication infrastructure for multiple FPGA platforms [35]. It is possible to generalize either scheme to handle multiple computational domains for arbitrary physical topologies.

4.6 Implementing the IFFT Module in BCL

Our partitioned example now assumes that the IFFT functionality will run in hardware, but we have not yet shown how BCL can be used to design efficient HW. There is substantial literature on this topic, but here we provide a short discussion in terms of IFFT which also illustrates the tradeoffs between hardware and software quality. Though the IFFT interface was specified to operate on dynamic sizes, the resulting mechanism needlessly obscures the discussion. Therefore, we present the subsequent designs assuming a fixed frame size of sixty-four, though the reasoning applies equally to the fully flexible design.

Unpipelined: Perhaps the most natural description of the IFFT is the nested **for** loop contained within a single rule, shown below:

```
module mkIFFTComb(IFFT#(k, t))
  FIFO#(Vector#(k, t))  inQ <- mkFIFO()
  FIFO#(Vector#(k, t)) outQ <- mkFIFO()

  method Action input(Vector#(k,t) x)
    inQ.enq(x)

  rule doIFFT
    let x = inQ.first(); inQ.deq();
    for(int stage = 0; stage < 3; stage++)
      for(int pos = 0; pos < 16; pos++)
        x = applyRadix(stage, pos, x);
    outQ.enq(x);

  method Vector#(k,t)) output()
   return outQ.first();

  method Action deq()
   outQ.deq();
endmodule
```

In this module, all the computation is done by the "doIFFT" rule, but to correctly implement the asynchronous interface, two FIFOs are used to buffer values. In software, the rule will compile into efficient loops, while in hardware they will be fully unrolled into a single block of combinational logic. Anyone with experience in hardware design will recognize that this circuit will produce an extremely long combinational path which will need to be clocked very slowly.

Pipelined: A better implementation would compute each stage of the IFFT in separate cycles. This cuts the critical path in hardware and introduces interesting pipelining opportunities. To do this we need to add some extra buffering in the form of additional FIFOs:

```
module mkIFFTPipe(IFFT#(k, t))
  Vector#(4,FIFO#(Vector#(k, t))) buff <- replM(mkFIFO);

  method Action input(Vector#(k,t) x)
    buff[0].enq(x);

  for(stage = 0; stage < 3; stage++) // gen. the rules
    rule stageN
      let x = buff[stage].first(); buff[stage].deq();
      for(int pos = 0; pos < 16; pos++)
        x = applyRadix(0, pos, x);
      buff[stage+1].enq(x);

  method Vector#(k,t) output()
   return buff[3].first();

  method Action deq()
   buff[3].deq();
endmodule
```

The for-loop which encloses the rule "stageN" is meta-syntax for writing three rules manually. These rules are generated by unfolding the loop. Consequently, in this module each stage is represented by a single rule which can fire independently. The implicit conditions (formed by the internal conditions on the FIFOs) in each stage rule enforces the necessary data dependencies to guarantee correct results are generated.

When compiling "mkIFFTPipe" to software we want to exploit data locality and process a single frame at a time, a task which can be accomplished by executing each IFFT

stage in dataflow order. This reconstruction of the original outer-level loop and has the effect of "passing the algorithm over the data". In hardware, on the other hand, we want to exploit pipeline parallelism and in each clock cycle run each rule once on different data, an approach which can be characterized as "passing the data through the algorithm". There are a host of other variations on this pipeline which are easily specified in BCL which we do not discuss for the sake of brevity. It should be noted that as the implementations become more "hardware-friendly" the task of synthesizing efficient software often becomes increasingly challenging.

4.7 Interfacing with The Software Stack

As discussed in Chapter 3, BCL can be used to specify entire programs or it can be used to specify nodes in the high-level dataflow graphs. The integration of the interfaces is straightforward, since get/put interfaces which of the FIFO dataflow edges have a one-to-one correspondence with the synchronizing FIFOs discussed earlier in this chapter. The external input/output FIFO edges are exposed as just another flavor of mkSyncFIFO.

The final question of integration revolves around who owns "main". In HW, a BCL-generated partition can be compiled as a separate program with its own Esposito [34] scheduler because it is insulated from all other hardware nodes by the synchronizer FIFOs. In SW, we assume that the BCL-generated nodes do not own main, so it is incumbent on the programmer to invoke the schedules of the BCL-generated partitions to ensure forward progress. With sufficient insight into the rule structure, the programmer could invoke the methods implementing the BCL rules directly, resulting in a much more efficient schedule.

Chapter 5

Implementing BCL

Shown in Figure 5-1 is a block diagram of a BCL Compiler whose implementation is described in this chapter. Starting with a BCL specification, the compiler generates a C++ implementation of the software partition and a BSV implementation of the hardware partition. The generated C++ is further compiled (using g++) with the appropriate library code to an executable which can be executed on supported platforms. The Hardware partitions are compiled to Verilog using bsc from Bluespec Inc., which is then synthesized using the Xilinx tool-chain and downloaded to the FPGA. Supported platforms consist of systems with a tightly integrated FPGA and microprocessor, for which an abstract model of the communication fabric has been created for use by the compiler. In addition, libraries in both hardware and software must be implemented to provide a standardized (albeit low-level) interface to the communication fabric for use by the generated code.

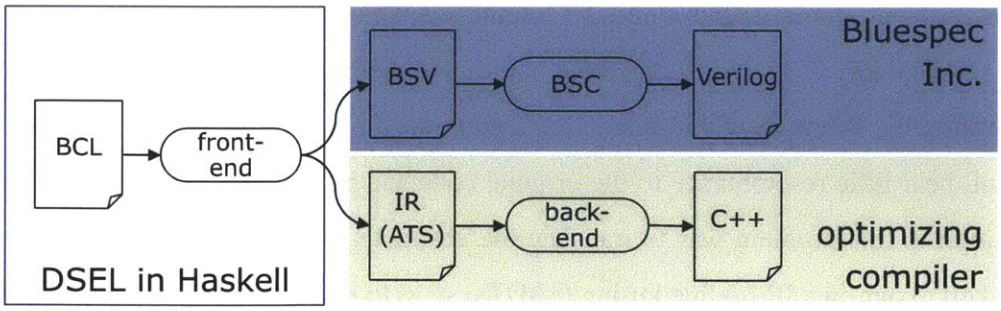


Figure 5-1: Organization of the BCL Compiler

5.1 Embedding BCL in Haskell

To avoid the drudgery of implementing a compiler front-end, we decided to implement the BCL language as a deep embedding in Haskell. This enables the BCL programmer to describe the module structure using all the powerful language features he is accustomed to when writing pure Haskell. In this sense, Haskell is being used as a meta-programming language in which a programmer describes a design whose dynamic behavior is fully captured by the formal semantics given in Chapter 4.

The BCL back-end has been implemented as a library in Haskell, so once the design has been specified, the programmer compiles his design using GHC. Running the resulting executable will output a C++ implementation of the software partition and a BSV implementation of the hardware partition. While not central to this thesis, the details of this embedding are interesting enough to merit some discussion. What follows is a discussion of a few of the more interesting aspects of the embedding.

Template Haskell for Preserving Names: One approach to specifying BCL program structure is to directly define the functions in Haskell which can be composed by the programmer to form rules and methods. For example, we could define an ActionValue monad similar to the Haskell State monad and then define a function which takes as arguments a register name and a value and updates register inside the monad. As long as the programmer restricts himself to simple expressions, "unwinding" the monad in GHC will reveal a structure very similar to the original source code. In addition, the C++ code to implement this monadic structure will resemble the original Haskell source quite closely. On the other hand, if the programmer introduces *let*-bindings and more complex expressions to simplify the source code, GHC will remove the names and modify the structure of the expressions substantially. While the C++ code to implement the monadic structure will still be correct, it will bear little resemblance to the original code and be far more difficult to debug. Our solution to this problem was to translate the abstract syntax trees generated by Template Haskell to our own IR (defined using GADTs) so as to restrict the kinds of transformations we allow to the BCL programs.

Template Haskell extends Haskell 98 with the ability to do type-safe meta program-

ming. With the introduction of an *unquoting* operator, a programmer can manipulate the Haskell language using Haskell. This is done by facilitating the conversation between concrete syntax and abstract syntax trees (AST), and because the ASTs are represented using Haskell data-types, they can be programatically manipulated by other Haskell routines. With a few restrictions, the programmer can refiy code segments, transform them, and splice them back into the original program. Alternately, the transformations can generate completely new code, which is what we do for BCL.

Phantom Types To Enforce Computational Domains: In addition to translating Template Haskell's ASTs to our own IR, we can further annotate the structures with additional type information pertaining to the computational domain. Phantom types in Haskell have type parameters which don't appear on the right-hand side of the definition and we use these in the implementation of the BCL IR to encode the computational domain.

Each expression in our IR contains a phantom type representing a computational domain. These types are then restricted during action and expression composition. For example, the constructor for a guarded action (*when*-action) in the IR takes as arguments a Boolean expression in computational domain n and an Action in computational domain n. The resulting action continues to be polymorphic in the domain type, but for the program to type check the types of the expression and the action must be consistent. If the *when*-action is used in a rule whose domain type has been restricted to *hw*, this will propagate through all the sub-actions and sub-expressions in that rule. If an expression references an interface method with an inconsistent type restriction, GHC will fail to compile the BCL program and return a type error.

Building an IR using GADTs: GADTs were added to GHC in 2005 [57], etc., and their use has been well documented. We have defined the IR for BCL using this technique because it drastically reduces the complexity of the translation to C++ or BSV since we can enforce the type constraints of the underlying expressions in the structure of the IR.

Implementing the ActionValue Monad: Expressions compose naturally, but Rules and Action methods are built up incrementally through the use of the action value monad. This monad is polymorphic in the compositional semantics (sequential or parallel) as well as

the return type. Since this lies at the heart of the BCL language, it was surprising how succinctly this behavior could be described.

5.2 Syntax Directed Compilation Scheme

Before generating C++ code we must canonicalize the structure of the IR in order to correctly handle the lazy evaluation semantics of BCL. Following the canonicalization phase, each BCL module can be compiled into a C++ class, where each of the module's rules and methods is compiled into a separate class method. During the course of translation we may need to create shadow copies of the state of an object. This is accomplished by constructing a new object of the same class using the copy constructor. The translation procedure maintains an environment, ρ, which maps BCL module instance names to the active class instance corresponding to that name. A new ρ containing all immediate submodules is constructed for the compilation of each BCL module, and serves as the *initial* environment for the translation of each internal rule and method. In this section, we present a detailed compilation scheme for BCL. We do so from the bottom up, starting with the compilation of expressions, then actions and rules, and concluding with modules. It should be noted that the scheme presented initially is far from optimal, and refinements to it are discussed in the following section.

For the sake of brevity we take a few notational liberties in describing translation rules. Generated C++ code is represented by the conjunction of three different idioms: literal C++ code (given in the `true text font`), syntactic objects which evaluate to yield C++ code (given in the document font), and environment variables used by the compiler procedures (represented as symbols). The names of compiler procedures that generate code fragments are given in **boldface**.

5.2.1 Canonicalizing

One approach to a C++ implementation of BCL would be to directly interpret the λ-calculus implementation [29]. While this is correct, it would not be very efficient since the top-level rule translation procedure computes the state updates and guards separately, updating

global state only at the end only if the union of the guards evaluates to $True$. A more efficient strategy would is to interleave the evaluation of the state updates and guards in such a way that we can abort the evaluation of the state updates once a guard failure is encountered. Unlike BCL, C++ has strict let-binding and function arguments, which complicates matters somewhat. We will need to transform the structure of the expressions to enable the more efficient "abort rule on guard failure" execution strategy we have just described. Consider the following BCL expression, which although it contains a **when** expression whose guard always evaluates to $False$, should only fail if the evaluation of p returns false (assuming the p contains no **when** expressions):

```
let x = when False 3
in if p then 5 else x
```

Since bound expressions in BCL have in-lining semantics (otherwise known as lazy), we need to modify the structure of the IR, since a direct translation of this expression to C++ will incorrectly *always* cause a guard failure. If BCL expressions permitted the infinite computation, a similar concern would arise for $\perp$. Instead of treating guards as pure data, we transform the IR to remove **when** clauses from let-bindings and value-method definitions. Intuitively, we lift all internal **when**s to the top-level using the axioms in Figure 4-6. Then we transform these bound top-level guards by splitting the binding into a body part and a guard part and moving the `when`s into the call sites. This can be done mechanically with the following procedure:

1. Replace all value method invocations `m.f(e)` with `m.`f_{Body}`(e) when m.`f_{Guard}`(e)`. Similarly replace any let-bound variable instances `x` whose value is a when expression, with the expression of two fresh variables `xb` **when** `xg`.

2. Lift all whens in expression to the top-level using the procedure $\mathbf{LW}_e$ in Figure 5-2. After this, all guards exist only in top-level expressions (*i.e.,* those which are not directly in subexpressions, only actions and method bindings).

3. Replace all value method definitions `m.f = λ x. e when g` with the two method definitions:

- m.f_{Body} = λ x.e
- m.f_{Guard} = λ x.g

4. Similarly convert all let bindings of variable x to define the two fresh variables. The variables defined here were used but in Step 1:
   - x = eb when eg in e $\Rightarrow$ xb = eb in xg = eg in e

After canonicalizing the structure of the IR, the translation to C++ can proceed using the syntax-directed compilation procedures which follow.

5.2.2 Compiling Expressions

The translation of a BCL expression produces a C++ expression and one or more statements that must be executed before the expression. These additional statements are responsible for executing parts of the expression that do not have directly corresponding representations in C++ expressions such as let bindings. The procedure to translate expressions (**TE**) is shown in Figure 5-3; it is straightforward because expressions are completely functional. The only clause needing further explanation is the guarded expression (e **when** ew). Upon evaluation, the failure of a sub-expression's guard should cause the entire expression to evaluate to $\perp$. If the value of an expression does evaluate to $\perp$, then its use in an action causes that action to have *no effect*. The "throw" in case of guard failure is "caught" in the lexically outermost action enclosing the expression.

5.2.3 Compiling Actions

A rule is composed of actions and any of these actions can be guarded. Earlier we explained the meaning of a guarded action by saying that a rule is not eligible to fire (execute) unless its guard evaluates to true. However, due to conditional and sequential composition of actions, in general it is impossible to know if the guards of all the constituent actions of a rule are true before we execute the rule. To circumvent this limitation, we execute a rule in three phases: In the first phase we create a shadow of all the state elements using

$$
\begin{array}{rl}
\mathbf{LW}_e\ [\![\ \mathrm{r}\ ]\!] = & (r\ \textbf{when}\ \texttt{true}) \\
\\
\mathbf{LW}_e\ [\![\ \mathrm{c}\ ]\!] = & (c\ \textbf{when}\ \texttt{true}) \\
\\
\mathbf{LW}_e\ [\![\ \mathrm{t}\ ]\!] = & (t\ \textbf{when}\ \texttt{true}) \\
\\
\mathbf{LW}_e\ [\![\ e_1\ \ \mathrm{op}\ \ e_2\ ]\!] = & (e'_1\ op\ e'_2)\ \textbf{when}\ (e_{1g} \wedge e_{2g}) \\
\text{where} & (e'_1\ \textbf{when}\ e'_{1g}) = \mathbf{LW}_e\ [\![\ e_1\ ]\!] \\
& (e'_2\ \textbf{when}\ e'_{2g}) = \mathbf{LW}_e\ [\![\ e_2\ ]\!] \\
\\
\mathbf{LW}_e\ [\![\ e_1\ \textbf{?}\ e2 : e3\ ]\!] = & (e'_1\ \textbf{?}\ e'_2 : e'_3)\ \textbf{when} \\
& (e_{1g} \wedge (e'_1\ \textbf{?}\ e_{2g} : e_{3g})) \\
\text{where} & (e'_1\ \textbf{when}\ e_{1g}) = \mathbf{LW}_e\ [\![\ e_1\ ]\!] \\
& (e'_2\ \textbf{when}\ e_{2g}) = \mathbf{LW}_e\ [\![\ e_2\ ]\!] \\
& (e'_3\ \textbf{when}\ e_{3g}) = \mathbf{LW}_e\ [\![\ e_3\ ]\!] \\
\\
\mathbf{LW}_e\ [\![\ e_1\ \textbf{when}\ e_2\ ]\!] = & e'_1\ \textbf{when}\ (e'_2 \wedge e_{1g} \wedge e_{2g}) \\
\text{where} & (e'_1\ \textbf{when}\ e_{1g}) = \mathbf{LW}_e\ [\![\ e_1\ ]\!] \\
& (e'_2\ \textbf{when}\ e_{2g}) = \mathbf{LW}_e\ [\![\ e_2\ ]\!] \\
\\
\mathbf{LW}_e\ [\![\ t = e_1\ \textbf{in}\ e_2\ ]\!] = & ((t' = e'_1)\ ;\ e'_2))\ \textbf{when} \\
& ((t' = e'_1)\ ;\ (t_g = e_{1g})\ ;\ e_{2g}] \\
\text{where} & (e'_1\ \textbf{when}\ e_{1g}) = \mathbf{LW}_e\ [\![\ e_1\ ]\!] \\
& e_3 = e_2[(t'\ \textbf{when}\ t_g)/t] \\
& (e'_2\ \textbf{when}\ e_{2g}) = \mathbf{LW}_e\ [\![\ e_3\ ]\!] \\
\\
\mathbf{LW}_e\ [\![\ m.f(e)\ ]\!] = & m.f(e')\ \textbf{when}\ e_g \\
\text{where} & (e'\ \textbf{when}\ e_g) = \mathbf{LW}_e\ [\![\ e\ ]\!]
\end{array}
$$

Figure 5-2: Procedure to implement **when** lifting in BCL Expressions. If method calls and bound variables are split following the algorithm given in 5.2.1, this procedure will result in guard-free expressions.

the `copy` constructor. We then execute all constituent actions, updating the shadow state. Sometimes more shadows are needed to support the parallel semantics within an action. (In our translation rules, we only copy and create new environments and let the generated C++ code mutate the objects referenced by the environment). Finally, if no guard failures are encountered we commit the shadows, that is, atomically update the real state variables with values of the shadowed state variables using `parMerge` or `seqMerge` depending upon the action composition semantics. On the other hand if the evaluation encounters a failed guard, it aborts the computation and the state variables are not updated.

```
TE :: Env × [[ e ]] → ( CStmt, CExpr )

TE ρ [[ r ]] = (;, ρ[r].read())
TE ρ [[ c ]] = (;, c)
TE ρ [[ t ]] = (;, t)

TE ρ [[ e1 op e2 ]] =   (s1;s2, ce1 op ce2)
              where   (s1, ce1) =  TE ρ [[ e1 ]]
                      (s2, ce2) =  TE ρ [[ e2 ]]

TE ρ [[ ep ? et : ef ]] =   (sp; st; sf, cep ? cet : cef)
                  where   (sp, cep) =  TE ρ [[ ep ]]
                           (st, cet) =  TE ρ [[ et ]]
                           (sf, cef) =  TE ρ [[ ef ]]

TE ρ [[ e when ew ]] =   (se; sw; if(!cw){throw GuardFail;},ce)
                  where    (se, ce) =  TE ρ [[ e ]]
                          (sw, cw) =  TE ρ [[ ew ]]

TE ρ [[ t = et in eb ]] =   (st; t = ct; sb, cb)
                  where    (st, ct) =  TE ρ [[ et ]]
                          (sb, cb) =  TE ρ [[ e ]]

TE ρ [[ m.f(e) ]] =   (se, ρ[m].f(ce))
              where   (se, ce) =  TE ρ [[ e ]]
```

Figure 5-3: Translation of Expressions

For perspicuity, the rules present a slightly inefficient translation where shadows of the entire environment are created whenever a shadow may be needed. Figure 5-4 gives the procedure for translating BCL actions (**TA**).

State Assignment ($r := e$)**:** This causes a side-effect in the relevant part of the state of the object which can be extracted from ρ. If e evaluates to bottom, the control would have already been transferred automatically up the call stack via the `throw` in e.

Parallel Composition ($a1 \mid a2$)**:** Both $a1$ and $a2$ observe the same initial state, though they update the state separately. Consider the parallel actions $r_1 := r_2 | r_2 := r_1$ which swap the values of r_1 and r_2. Such semantics are naturally implemented in hardware as swaps can be done with no intermediate state (the values are read in the beginning of a clock cycle

```
TA :: Env × ⟦ a ⟧ → CStmt

TA ρ ⟦ r := e ⟧ =   se; ρ[r].write(ce);
          where   (se, ce) =   TE ρ ⟦ e ⟧

TA ρ ⟦ if e then a ⟧ =   se; if(ce){TA ρ ⟦ a ⟧}
                where   (se, ce) =   TE ρ ⟦ e ⟧

TA ρ ⟦ a1 | a2 ⟧ = cs1; cs2; (TA ρ1 ⟦ a1 ⟧); (TA ρ2 ⟦ a2 ⟧); pm; ms;
          where (cs1, ρ1) =   makeShadow ρ
                (cs2, ρ2) =   makeShadow ρ
                (pm, ρ3) =   unifyParShadows ρ1 ρ2
                     ms =   commitShadow ρ  ρ3

TA ρ ⟦ a1;a2 ⟧ = cs; (TA ρ1 ⟦ a1 ⟧); (TA ρ1 ⟦ a2 ⟧); ms;
        where (cs, ρ1) =   makeShadow ρ
                   ms =   commitShadow ρρ1

TA ρ ⟦  a when e  ⟧ =   se;if(!ce){throw GuardFail;};ca
                where   (se, ce) =   TE ρ ⟦ e ⟧
                             ca =   TA ρ ⟦ a ⟧

TA ρ ⟦  t = e in a  ⟧ =   se; t  =  ce; (TA ρ ⟦ a ⟧)
                where   (se, ce)   =   TE ρ ⟦ e ⟧

TA ρ ⟦ m.g(e) ⟧ =   se; (ρ[m].g(ce));
            where   (se, ce) =   TE ρ ⟦ e ⟧

TA ρ ⟦ loop e a ⟧ =   cs;  while(true){se;
                      if(!ce)  break;  ca;}  ms;
              where   (cs,ρ1) =   makeShadow ρ
                      (se, ce) =   TE ρ ⟦ e ⟧
                          ms =   commitShadow ρ ρ1
                          ca =   TA ρ1 ⟦ a ⟧

TA ρ ⟦ loopGuard e a ⟧ =   try{while(true){cs;se;if(!ce){break;}ca;ms;
                           }catch{}
                     where   (cs,ρ1) =   makeShadow ρ
                             (se, ce) =   TE ρ ⟦ e ⟧
                                 ms =   commitShadow ρ ρ1
                                 ca =   TA ρ1 ⟦ a ⟧;
```

Figure 5-4: Translation of Actions

and updated at the end of it). However, in software if we update r_1 before executing the second action, then the second action will read the new value for r_1 instead of the old one. To avoid this problem, the compiler creates shadow states for each parallel action, which are subsequently merged after both actions have executed without guard failures. In a legal program, the updates of parallel actions must be to disjoint state elements. Violation of this condition can only be detected dynamically, in which case an error is thrown.

The compiler uses several procedures to generate code to be used in implementing parallel composition. The **makeShadow** procedure takes as its argument an environment (ρ) and returns a tuple consisting of a new environment (say $\rho 1$), and C++ statements (say cs1). cs1 is executed to declare and initialize the state elements referenced to in $\rho 1$. The new environments are then used in the translation of each of the actions. The procedure **unifyParShadows** is used to unify $\rho 1$ and $\rho 2$, implicitly checking for consistency. Along with $\rho 3$, which contains the names of the unified state elements, it returns a C++ statement (pm) which actually implements the unification. Lastly, the **commitShadow** procedure generates code (ms) to commit the speculative state held in $\rho 3$ back into the original state ρ. In order to understand **unifyParShdows**, consider the parallel merge of two primitive Registers, each of which track their own modification with a dirty bit:

```
void parMerge(Reg<T>& r1, Reg<T>& r2){
  if (r1.dirty && r2.dirty) {
    throw ParMergeError;
  }
  if (r2.dirty) {
    r1.val = r2.read();
    r1.dirty = true;
}}
```

From here, the reader can extrapolate the implementation to other primitive modules such as the `VectorReg` used in the IFFT implementation. The `parMerge` for this module would most likely allow updates to disjoint locations in parallel branches of execution, but throw an error if the same location were written. **unifyParShadows** generates code which recursively invokes `parMerge` point-wise on the two environments, whereas **commitShadow** ρ $\rho 1$ simply performs point-wise updates of objects in ρ from dirty objects in

$\rho 1$ by invoking `seqMerge`.

Sequential composition ($a1; a2$)**:** The action $a1; a2$ represents the execution of $a1$ followed by $a2$. $a2$ observes the full effect of $a1$, but due to the atomic nature of action composition, no one can observe $a1$'s updates without also observing $a2$'s. The subtlety in sequential composition is that if $a1$ succeeds but $a2$ fails, we need some way of undoing $a1$'s updates. Its because of this that we need to create a shadow state before executing either action. As the sequential rule in Figure 5-4 shows, after creating the shadow $\rho 1$, we pass it to the translation of both $a1$ and $a2$. The code block ms commits the resulting state. However $a1; a2; a3$ can all be executed using only one shadow.

Guarded Action (a **when** e)**:** The **when** keyword allows the programmer to guard any action. The C++ translation of guarded actions executes the guard, se, followed by the expression ce, and throws a guardFail exception if ce evaluates to false. After this, the code implementing a is executed.

Atomic Loop (loop e a)**:** The semantics of this loop can be understood as the reduction: **loop** e a $\Rightarrow$ **if** e **then** $(a;$ **loop** e $a)$ **else** $noAction$. In other words all iterations are sequentially composed to form one atomic action. Consequently, if any iteration fails, the whole loop action fails. This can be implemented using one shadow which is created before entering the loop.

Protected Loop (loopGuard e a)**:** The difference between the protected loop and the atomic loop are in the termination semantics. In protected loop, if a guard in a loop iteration fails, that *iteration* is treated as `noAction`. In the atomic loop, the entire loop action becomes `noAction`.

The reader might notice the absence of a parallel loop. This is not due to any fundamental barrier in the semantics, but rather the authors' prejudice in how this affects design patterns in BCL.

5.2.4 Compiling Rules and Methods

Figure 5-5 gives the translation of rules and methods, differentiating between action and value methods. The important thing to note is that rules catch guard failures, whereas

methods do not. This is consistent with our implementation of the BCL semantics that specify rules as *top level* objects which cannot be invoked within the language. Methods, on the other hand, must be called from inside a rule or another method.

genRule ρ ⟦ (Rule name a) ⟧ =
  `void name(){try{`**TA** ρ ⟦ a ⟧`}catch{guardFail};}`

genAMeth ρ ⟦ (AMeth name v a) ⟧ = `void name(t v){`**TA** ρ ⟦ a ⟧`}`

genVMeth ρ ⟦ (VMeth name v e) ⟧ = let (se,ce) = **TE** ρ ⟦ e ⟧
  in `void name(t v){se;return ce;}`

Figure 5-5: Translation of Rules and Methods

When generating code for rules and methods, we invoke the compiler rules **TA** and **TE**, which in turn make use of additional rules to generate code for the creation and merging (in either parallel or sequential contexts) of shadows. These rules, discussed informally in the previous section, are presented in Figure 5-6.

Env :: BCLExpr → CExpr

makeShadow :: **Env** → ([CStmt],**Env**)
makeShadow ρ = (copy_stmts, new_mapping)
  where **sh** ⟦ e↦n ⟧ = (new `t = n.copy()`,[e↦`t`])
        (copy_stmts, new_mapping) = unzip (map **sh** ρ)

commitShadow :: **Env** × **Env** → [CStmt]
commitShadow ρ1 ρ2 =
  map (λ(⟦ e↦n ⟧). `e.seqMerge(`ρ2[n]`)`) ρ1

unifyParShadows :: **Env** × **Env** → ([CStmt],**Env**)
unifyParShadows ρ1 ρ2 = (merge_stmts, new_maping)
  where **sh** [n↦n] = (new `t = n.parMerge(`ρ2[e]`)`, [e↦`t`])
        (merge_stmts, new_mapping) = unzip (map **sh** ρ1)

Figure 5-6: Production Rules

5.2.5 Compiling Modules

The C++ class corresponding to a BCL module has five methods in addition to a method for each module's rules and methods. These five methods are a default constructor (used to instantiate its submodules recursively), a copy constructor (used to generate shadows, also recursive), `parMerge`, `seqMerge`, and a method to execute internal rules. The translation procedure for modules is given in Figure 5-7. Figure 5-8 gives the definition of a register primitive, from which the reader can extrapolate how other primitive state might be implemented.

```
TM :: [[ m ]] → CClassDef
TM [[ (ModuleDef name args insts rules ameths vmeths) ]] =
class name {
  public:
    map (λ[[ (Inst mn n _) ]]. mn* n;) insts

    name () {map(λ[[ (Inst mn n v) ]]. n=new  mn (v) ; ) insts

    name* copy () {
      name rv = new name;
      map (λ[[ (Inst mn n _) ]]. rv.n = n.copy () ;)  insts
      return rv;
    }

    ~name () {map (λ[[ (Inst _ n _) ]].delete  n; ) insts}

    void ParMerge (name shadow) {
      map (λ[[ (Inst mn n _) ]]. n.ParMerge (shadow.n)) insts
    }

    void SeqMerge (name shadow) {
      map (λ[[ (Inst mn n _) ]]. n.ModuleMerge (shadow.n)) insts
    }

    map genRule rules
    map genAMeth ameths
    map genVMeth vmeths
    void execSchedule () {
      map (λ[[ (Rule n a) ]].n () ; ) rules
      map (λ[[ (Inst mn n _) ]].n.execSchedule () ; ) insts}
}
```

Figure 5-7: Translation of Modules Definitions to C++ Class Definition

```
template<typename T> class Reg{
  public:
    bool    modified;
    T          value;
    Reg<T>  *parent;
    inline  T& read(){return modified ? value : parent->read();}
    inline void write(const T& new_val){
      modified = true;
      value = new_val;}
    Reg<T>(T def):
      modified(true),
      value(def),
      parent(NULL) {}
    Reg<T>(class Reg<T>& ref):
      modified(false),
      parent(&ref) {}
    ~Reg<T>() {value.~T();}
    inline void ModuleMerge(class Reg<T>& a){
      modified |= a.modified;
      if (a.modified){value = a.value; }}
    inline void ParMerge(class Reg<T>& a){
      if(a.modified){
        if(modified){throw ParMergeFail;}
        value = a.value;
        modified = true;
    }}
    inline void execSchedule(){}
};
```

Figure 5-8: C++ Implementation of the BCL Register primitive. A Pointer to the parent object is used to avoid unnecessary copying of large values.

Each BCL module instantiates all of its submodules and specifies their initial state. The corresponding C++ default constructor implements this behavior exactly. Thus, by instantiating the top module, the entire program state is recursively instantiated. For the translation of the methods and rules of each module definition, an environment ρ is constructed to refer to these newly instantiated objects. This environment is used to compile all rules and methods of this module as shown in Figure 5-5.

The execution semantics of a BCL program is defined by the firing of its constituent rules. While each rule modifies the state deterministically, nondeterminism is introduced via the choice of which ready rule to execute. The range of behaviors that a collection of

rules and state can produce is described in Figure 4-2.

This nondeterminism is resolved in each module by its implementation of the method `execSchedule`. There are many different scheduling choices which greatly impact execution performance. The reference implementation shown in Figure 5-7 simply iterates through each rule in a module, before recursively invoking `execSchedule` on all submodules one by one.

Implementing `main`. A `main()` function is required to drive the execution of the program. With a local schedule defined for each C++ class implementing a BCL module, this task is trivial, as shown in Figure 5-9. Of course, the choice of schedule is one of the most important ways in which we can optimize the behavior of BCL implementations, and we will discuss more sophisticated approaches to scheduling in Chapter 6.

```
void main(){
  TopMod* t = new TopMod();
  while(true){
    t->execSchedule();
  }
}
```

Figure 5-9: Reference C++ implementation of `main()`, for a BCL program whose top-level module is of type 'TopMod'

Since BCL is not sufficiently dynamic to describe software at higher levels in the software stack, facilities have to be provided to interface BCL with software implemented directly in C++ (for example). The BCL compiler can generate an interface of guarded methods, and an application seeking to use BCL as a slave can directly call these methods after first verifying that a method's guard is "ready". If a peer-oriented view is desired, the BCL design may be required to call directly into the external application. This requires designers to present an appropriate guarded interface to the BCL application. In either case, the BCL schedule must be somehow invoked. If the interfaces between the BCL-generated C++ and the remainder of the SW stack are implemented using IPC synchronization primitives, a separate thread can be dedicated to execute the rule schedule in support of the peer-oriented view.

5.3 Optimizations

In an effort to make it more easily understandable, the syntax-directed translation scheme given in the previous section is substantially simpler than the code actually generated by our compiler. In this section, we roll back some of these simplifications and refine parts of the SDC to more accurately reflect the code generated by the BCL compiler.

5.3.1 Reducing the Cost of Shadow States

At a high level, the software implementation of BCL can be explained by pointing out the differences between rule-based systems and Transactional Memory. Both these systems have the *linearizability* property, that is, the semantics of the system are understandable by executing rules (transactions) in some sequential order. However, the implementation techniques for the two systems are quite different. In TM systems, multiple transactions are executed eagerly, and all except one are aborted if a conflict is detected. Upon failure, TM systems transfer control back to the transaction's boundary, and the transaction is re-executed. In TM systems, conflicts are typically defined in terms of read/write conflicts, though more sophisticated schemes exist. The challenge in the implementation of TM systems is to find low-cost mechanisms for detecting conflicting transactions and rolling back the computation in case of an aborted transaction. The possibility of a rollback invariably requires shadow states or check-points.

The implementation of rule-based systems such as BCL, on the other hand, there is freedom to pre-schedule rules in such a way that conflicting rules are never executed simultaneously. Thus, there is no need for dynamic conflict detection. In the approach employed by bsc, many rule guards are evaluated in parallel, and among the rules whose guards are *true*, all non-conflicting rules are scheduled to be executed simultaneously. The compiler does pair-wise static analysis to conservatively estimate conflicts between rules. Thus, compilers for rule-based systems construct a scheduler for each design using the static conflicts, as well as dynamic information about which rules are ready to *fire*.

In our BCL implementation we never simultaneously schedule conflicting rules, in fact it is a single threaded implementation which *never* attempts concurrent execution of rules

even when they do not conflict. In spite of this our implementation still requires shadow states. In simple designs all guards can be lifted out of the rule, the scheduler can guarantee that a rule will not fail before evaluating its body. In complex rules, not all guards can be lifted, which leaves the possibility of internal guard failure. This possibility of failure necessitates shadows in our system, providing an incentive to reduce their cost.

Here we introduce the first modification to the original SDC. For each action we must construct a new state object, however this construction is expensive and requiring a copy of the entire state as shown in Figure 5-4 is wasteful. We can statically analyze the action body, and create a list of sub-modules which are modified in that action. Copying only these modules will always be less expensive than the entire state. Pointers to state elements which are read but not modified in the action are added to ρ to complete the mapping. If we can delay the shadow creation by executing the copy of each object immediately before it's first update (a lazy approach to shadow creation), we can further reduce the overall cost of shadowing since control-flow in the rule body might let us avoid the copy entirely.

Incomplete guard lifting is only one reason we require shadow states in our BCL implementation. There is another instance which arises from the internal concurrency expressed through the use of parallel action composition with a rule body. In general, "(A1 | A2)" will require creating two shadows of the incoming state to execute 'A1' and 'A2' in isolation. After the execution of 'A1' and 'A2', the copies will have to be merged into a single shadow. This merge may produce a dynamic error if 'A1' and 'A2' modify the same state element; such actions are illegal. The following example illustrates this, where 'c1' and 'c2' are dynamic boolean expressions, 'f' is a FIFO, and 'a' and 'b' are registers:

```
(if c1 then a := f.first(); f.deq()) |
(if c2 then b := f.first(); f.deq())
```

If 'c1' and 'c2' both evaluate to true and 'f' is not empty then both the sub-actions will attempt to dequeue the same element, which is an error. Even if there is no danger of parallel merge errors, shadow state is required, as illustrated by the following simple example, which swaps the values of two registers: "a := b | b := a". Optimizing the shadow creation for the parallel connective is similar to the lazy approach described earlier.

Sequentialization of Parallel Actions: The cost of shadowing for parallel actions can be reduced by transforming them into an equivalent sequential composition of actions. For example, (A | B) is equivalent to (A ; B) if the intersection of the write-set of action A and the read-set of action B is empty. The equivalence of (A | B) and (B | A) gives us even more flexibility in finding an equivalent sequential form. In addition, action methods invoked on the same module but in parallel branches can be invoked in sequence if they modify disjoint internal states. In both cases, since the two actions were originally composed in parallel, their respective guards may be lifted, giving us a sequential composition *and* the ability to guarantee that the second action will complete without a guard failure. Since parallel composition is most naturally expressed in hardware and sequential composition in software, recognizing these patterns is important when implementing a "hardware" module in software. Sometimes sequentialization of parallel actions is made possible by introducing some shadow state (consider the swap example). Even this turns out to be a win because static allocation of state is more efficient than *dynamic* allocation.

5.3.2 Reducing the Cost of Guard Failures

Guard Lifting: When executing a rule whose guard might fail, it is preferable that it happen as early in the execution as possible, since early failure avoids the useless execution of the remainder of the rule body. Consider the following transformation of a BCL expression:

$$(\text{A1 when A1}_g \mid \text{A2 when A2}_g) \Rightarrow (\text{A1} \mid \text{A2}) \text{ when } (\text{A1}_g \wedge \text{A2}_g)$$

By rules given in [25], it is possible to lift most guards to the top level. In some cases, we can transform rules to the form "A **when** E" where A and E are themselves guard-free. In this form, we can guarantee that if E evaluates to 'True', A will execute without any guard failures. With this knowledge, we can avoid the cost of using a try/catch block to detect guard failures in sub-module method invocations, and also perform the computation *in situ* to avoid the cost of commit entirely. The when-axioms (Figure 4-6) state that guards cannot be lifted through the sequential composition of actions or loops, which is why we cannot do away with state shadowing all together. A Procedure to implement guard lifting in actions is given in Figure 5-10.

$$
\begin{array}{rl}
LW_a[\![r := e]\!] = & (r := e') \textbf{ when } e_g \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![a \textbf{ when } e]\!] = & a' \textbf{ when } (a_g \wedge e' \wedge e_g) \\
\text{where} & (a' \textbf{ when } a_g) = LW_a[\![a]\!] \\
& (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![\textbf{if } e \textbf{ then } a]\!] = & (\textbf{if } e' \textbf{ then } a') \textbf{ when } (e_g \wedge (a_g \vee \neg e')) \\
\text{where} & (a' \textbf{ when } a_g) = LW_a[\![a]\!] \\
& (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![a_1 \mid a_2]\!] = & (a'_1 \mid a'_2) \textbf{ when } (a_{1g} \wedge a_{2g}) \\
\text{where} & (a'_1 \textbf{ when } a_{1g}) = LW_a[\![a_1]\!] \\
& (a'_2 \textbf{ when } a_{2g}) = LW_a[\![a_2]\!] \\
\\
LW_a[\![a_1 \,;\, a_2]\!] = & (a'_1 \,;\, LW_a[\![a_2]\!] \textbf{ when } a_{1g} \\
\text{where} & (a'_1 \textbf{ when } a_{1g}) = LW_a[\![a_1]\!] \\
\\
LW_a[\![t = e \textbf{ in } a]\!] = & ((t' = e') \textbf{ in } a') \textbf{ when } ((t' = e') \textbf{ in } (t_g = e_g) \textbf{ in } a_g) \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
& a_2 = a[(t' \textbf{ when } t_g)/t] \\
& (a' \textbf{ when } a_g) = LW_a[\![a_2]\!] \\
\\
LW_a[\![m.g(e)]\!] = & (m.g_b(e') \textbf{ when } e_g \wedge m.g_g(e') \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![\textbf{ loop } e\ a]\!] = & \textbf{loop } LW_e[\![e]\!]\ LW_a[\![a]\!] \\
\\
LW_a[\![\textbf{ localGuard } a]\!] = & \textbf{localGuard} LW_a[\![a]\!]
\end{array}
$$

Figure 5-10: Procedure to apply **when**-lifting to actions. $\mathbf{LW}_e$ is defined in Figure 5-2. The splitting of value-method calls and bound variables into body and guard components is assumed.

Avoiding Try/Catch: Even when guards cannot be lifted completely, we can still improve the quality of code when all methods in a rule are in-lined, or at least determine through static analysis that those remaining (methods on primitive modules) will not fail. In this case, we can dispense with the top-level try/catch block and instead handle all constituent **when** clauses explicitly, branching directly to the rollback code. Try/catch blocks in C++ are not cheap, and this analysis proves to be quite effective. To illustrate this optimization, consider the BCL rule below:

```
rule foo when (True) seq do
    a := 1
    f.enq(a)
    a := 0
```

The code generated without method in-lining is give in Figure 5-11, while the (far more efficient) code generated after method in-lining is given in Figure 5-12.

```
void foo(){
  FIFO _f(f);
  Reg _a(a);
  try{
    _a.write(0);
    _f.enq(_a.read());
    _a.write(1);
    f.commit(_f);
    a.commit(_a);
  }catch{
    //updates are dropped
  }
}
```

Figure 5-11: without in-lining

```
void foo(){
  FIFO _f(f);
  Reg _a(a);
  _a.write(0);
  if(_f.can_enq()){
    _f.enq(a_.read());
  else
    goto fail;
  _a.write(1);
  f.commit(_f);
  a.commit(_a);
fail:
}
```

Figure 5-12: with in-lining

5.3.3 Scheduling Decisions

In both hardware and software, the execution strategy has a great impact on performance. The most important concern in scheduling software is to choose a rule which will not fail, since partial execution of any rule is wasted work (the cost of partial execution and rollback). We are investigating the use of user-provided scheduling annotations to improve compiler analysis, but do not discuss it in this paper.

Parts of programs are sometimes identifiable as synchronous regions in which case well-known static scheduling techniques are applied [46]. When available, these synchronous guarantees can lead to very efficient code generation in both hardware and software implementations. Even without such synchronous guarantees, the compiler can exploit dataflow analysis which may reveal that the execution of one rule may enable another,

permitting the construction of longer sequences of rule invocations which successfully execute without guard failures, an important element in efficient software. This strategy is most effective when the program dataflow graph is sparse.

To illustrate the scheduling challenge, consider the following rule, which transfers a single audio frame between a producer module (p) and consumer module (c) non-atomically. If p empties or c is full before the complete frame has been transferred, the scheduler will attempt to complete the transfer during a subsequent rule invocation:

```
rule xferSW seq do
  cond_reg := true
  loop (cond_reg = true && cnt < frameSz) do
    cond_reg := false
    localGuard do
      cond_reg := true
      cnt := cnt+1
      let w = p.output();
      p.deq()
      c.input(w)
```

The sequential composition inherent in loops is not directly implementable in HW. For a hardware implementation, we might instead use a rule which transmits one word at a time, and rely on the scheduler to attempt to execute it once every HW clock cycle. The effects of the resulting non-atomic transfer of a single frame is identical, though the schedules are completely different:

```
rule xferHW when(cnt < frameSz) seq do
  let w = p.output();
  p.deq()
  c.input(w)
  cnt := cnt+1
```

Since one of our goals is to write code which can be moved between HW and SW with minimal changes, we need some way of resolving these two very different idioms. In essence, we would like to transfer a frame from p to c using `xferHW` with the same efficiency as if we had used `xferSW`. If the SW scheduler invokes `xferHW` in a loop, the overall performance of the transfer will not suffer. If the rules in c require a complete frame before

they are enabled, any attempts to execute those rules by the scheduler before `xferHW` has been invoked '`frameSZ`' times will be wasted effort. By employing completely different schedules, we are able to generate both efficient HW and SW from the same rules. A more in-depth discussion on this topic appears in Chapter 6.

5.3.4 Control Transfer

Except in the case of the parallel connective, we can avoid dynamic creation of shadow state all-together. A copy of the original state is allocated at program start-up time, and is referred to as the *persistent* shadow. This shadow is populated (in a change-log manner) as the rule executes. When a rule executes to completion, the shadow state is committed to the original program state. Consistency with the original state is maintained by executing a rollback if a rule fails. Unlike the shadows for rule execution, those required for parallel action execution are allocated and de-allocated dynamically. We chose this approach because after optimizations, parallel compositions are relatively infrequent. In Figures 5-13 and 5-14, we show how these persistent shadows are ultimately used in the code generated by the BCL compiler. Since the roll-back mechanism to maintain the coherence of the persistent shadow is far less expensive than dynamic object creation, this will result in more efficient code.

```
void foo(){
  try{
    a_s.write(0);
    f_s.enq(a_s.read());
    a_s.write(1);
    f.commit(f_s);
    a.commit(a_s);
  }catch{
    a_s.rollback(a);
    f_s.rollback(f);
  }
}
```

Figure 5-13: Adding Persistent Shadows to the code in Figure 5-11

```
void foo(){
  a_s.write(0);
  if(f_s.can_enq()){
    f_s.enq(a_s.read());
  else
    goto rollback;
  a_s.write(1);
  f.commit(f_s);
  a.commit(a_s);
rollback:
  a_s.rollback(a);
}
```

Figure 5-14: Adding Persistent Shadows to the code in Figure 5-12

5.4 Rules for Hardware vs. Rules for Software

One underlying principle of efficient software is that once data is loaded into the cache, it should be "processed completely" (whatever that means) before evicting it from the cache and loading more data. The degree to which the working set fits in the cache and spurious copying of data is avoided dictates the efficiency of the software implementation. In contrast, hardware implementations are characterized by streaming data from memory, processing it, and writing it back to memory without the need for a cache. The degree of pipelining and parallelism exposed by the microarchitecture will dictate the efficiency of the overall design. Moving (or copying) data from one pipeline stage to the next is required if we are to exploit the inherent parallelism in hardware. In less conventional terms, the goal of software is to pass the algorithm over the data, while hardware attempts to pass the data over the algorithm.

The promise of a language for HW-SW codesign is that once the program has been specified, different implementations each with their own HW-SW decomposition can be generated with minimal effort. The compiler presented in this chapter makes good on this promise, but there is a deeper question which we will attempt to illustrate through some examples. Even though the functionality can theoretically be implemented in either hardware or software, the rules may be written in such a way that they can be executed efficiently in software but not in hardware or vice versa.

5.4.1 IFFT: A Simple Pipeline

rule in Figure 5-15 directly implements an IFFT as a nested **for** loop. Some liberty with the notation is taken here, since the local variable 'x' gets reassigned each time the innermost loop is evaluated. One might expect us to use a register and write and read from it, but we currently have no hardware implementation for the sequential connective and we want to write rules which can be implemented as both HW *and* SW. Semantically, re-binding values to x is equivalent to introducing a new nested let-expression (scope) each time the rule body is evaluated. This rule is legal BCL, so let's consider the resulting hardware and software implementations

```
rule doIFFT par
  let x = inQ.first();
  inQ.deq();
  for(int stage = 0; stage < 3; stage++)
    for(int pos = 0; pos < 16; pos++)
      x = applyRadix(stage, pos, x);
  outQ.enq(x);
```

Figure 5-15: Rule Specifying IFFT for efficient SW implementation of a fixed-size IFFT

In software, we choose not to statically elaborate the **for** loops, and create a temporary variable to store the value of 'x'. The result is a *direct* implementation of the loops specified by the programmer. If we compile this rule to hardware, on the other hand, the structure will look significantly different, since the loops must be statically elaborated. The resulting structure is shown below:

```
rule doIFFT
  let x   = inQ.first(); inQ.deq();
  let x0  = applyRadix(0, 0, x)
  let x1  = applyRadix(0, 1, x0)
  ...
  let x10 = applyRadix(1, 0, xf)
  ...
  let x20 = applyRadix(2, 0, x1f)
  ...
  let x30 = applyRadix(3, 0, x2f)
  ...
  let x3f = applyRadix(3, f, x3e)
  outQ.enq(x3f);
```

While the elaborated rule will generate legal hardware, the long combinational paths generated by the BSV compiler will make its use in a larger design impractical due to the clock frequency limitations. This loop unrolling isn't as drastic as it might seem, since the inner loop (modulating the second argument of applyRadix) could have been specified using **foreach** if such an operator existed in BCL, and the resulting circuit will be only four applyRadix functions deep (this is still unacceptable, but is better than 64). A more sophisticated compilation has been proposed which permits rules such as this to be implemented

in multiple cycles [43]. The challenge with multi-cycle execution is that it must appear to be atomic with respect to all other rules in the system. This scheme has not been integrated into the BSV compiler, so until it is these types of rules will be highly impractical.

```
rule stage0
  let x = buff[0].first();
  buff[stag0].deq();
  for(int pos = 0; pos < 16; pos++)
    x = applyRadix(0, pos, x);
  buff[1].enq(x);

rule stage1
  let x = buff[1].first();
  buff[1].deq();
  for(int pos = 0; pos < 16; pos++)
    x = applyRadix(1, pos, x);
  buff[2].enq(x);

rule stage2
  let x = buff[2].first();
  buff[2].deq();
  for(int pos = 0; pos < 16; pos++)
    x = applyRadix(2, pos, x);
  buff[3].enq(x);

rule stage3
  let x = buff[3].first();
  buff[3].deq();
  for(int pos = 0; pos < 16; pos++)
    x = applyRadix(3, pos, x);
  outQ.enq(x);
```

Figure 5-16: Rule Specifying IFFT for efficient HW implementation of a fixed-size IFFT

If we knew that our eventual target was hardware, we would use the rules shown in Figure 5-16, which are very different from the rule in Figure 5-15. This design is different not only because we have assigned each stage its own rule, but because we have introduced buffering between each stage of the IFFT. As we mentioned in the context of the previous implementation, modulating the second argument to applyRadix will update disjoint elements of the 'x' vector. This means that the for a hardware implementation, the critical

path of each rule will be a single radix. Each stage of the IFFT is separated by a buffer and from this specification, bsc will generate a fully pipelined implementation, capable of producing one frame per cycle.

Attempting to create an efficient single-threaded software implementing from these "hardware-centric" rules is a challenge. If we assume the implementation resulting from the BCL code in Figure 5-15 to be our ultimate goal, can we achieve equivalent performance from our multi-rule system? Generating the code for each rule is simple and will produce four copies of the inner loop in our SW implementation, each with the stage argument to applyRadix hard-coded. The next challenge is to schedule the rules, and this is discussed at greater length in Chapter 6. Unless we implement a "perfect" schedule, there is no chance of achieving the ideal performance of the single SW-centric description, so for now let us assume an oracle which gives us the rule schedule [stage0,stage1,stage2,stage3]. If we invoke the rules in the order specified by the schedule, we will achieve the same ordered computation as the one-rule system. Because the computation is implemented in separate methods and additional buffering is introduced, achieving the same level of optimization in code generation will require inter-procedure optimization and a level of sophistication which is not implemented by most C++ compilers. Because of this, the performance difference between the SW generated from the BCL code in Figure 5-15 achieves almost 6 times the throughput of the SW generated from the BCL code in Figure 5-16. The difference in hardware performance between the two designs is approximately two orders of magnitude.

As this example illustrates, generating efficient software from rules involves much more than efficiently compiling the rule bodies. In spite of this challenge, we were able to achieve reasonable SW performance (assuming the scheduling oracle) from the hw-centric specification. Through the next example, we argue that achieving this parity automatically is impossible for an important set of designs.

5.4.2 Forney's Algorithm: A More Complex Pipeline

To further illustrate the distance between rules for efficient hardware and rules for efficient software, we examine alternate implementations of the error magnitude block from the

Reed-Solomon decoder discussed in Chapter 2. Using this example, Agarwal *et al.* [7] presented a range of designs to demonstrate the capability of C-to-gates technologies. They determined that no existing tool could infer an efficient hardware implementation from a straightforward C++ implementation of the algorithm. In order to achieve an efficient FPGA implementation with these tools, they concluded that one had to modify the C++ implementation to the point where it ultimately resembled the desired microarchitecture.

Shown in Figure 5-17 is a rule which implements the core of the Error Magnitude block in the Reed-Solomon pipeline. If the number of errors in a frame are below a certain threshold, Forney's algorithm is used to compute the roots of the error magnitude polynomial after which the error can be corrected. The flow control is quite simple: if the input is correct, pass it on to the output buffer, otherwise compute the polynomial and correct the value.

```
rule errorMag seq
  let x = inQ.first();
  let err = errorQ.first();
  inQ.deq();
  errorQ.deq();
  if (err) then do
      i.write(0);
      while(i.read < 17) do
        // iterative implementation of Forney's algorithm
        // final result is stored in register 't'
      outQ.enq(t.read);
   else outQ.enq(x);
```

Figure 5-17: SW-centric Rules Calculating the Error Magnitude if blocks in the Reed-Solomon Pipeline

The rule in Figure 5-17 will result in optimal software, but let us consider the quality of hardware which it will produce. Because the errorMag rule contains actions composed in sequence, we will need to transform it (either manually or automatically using well known techniques) to an FSM which the BCL hardware back-end (bsc) is able to compile to hardware. The resulting state machine will take either 1 or 17 cycles to process each token depending on whether the token contains an error. Since the number of errors (t) cannot

be greater than 16, a frame of length 223 (k) will require as many as 479 ($17t + (k - t)$) cycles.

In contrast to the SW-centric design in Figure 5-17, the rules in Figure 5-18 target efficient hardware. With sufficient buffering, we can achieve perfect concurrency between the error and noError pipelines. This overlapped execution, combined with the pipelined (as opposed to multi-cycle FSM) implementation of Forney's algorithm should give us a 2X speedup over the FSMs produced from the SW-centric rules shown previously. Because this module is on the critical path of the Reed-Solomon benchmark, this speedup is significant.

```
rule error0 when (errorQ.first()) par
  let x = inQ.first();
  inQ.deq();
  errorQ.deq();
  pq.enq(pos.read();
  pos.write(pos.read()+1);
  tx[0].enq(f0(x));

rule error1 par
  tx[1].enq(f1(tx[0].first));
  tx[0].deq;

...

rule errorF par
  outQ.enq(pq.first(), ff(tx[16].first()));
  tx[16].deq();
  pq.deq();


rule noError when (not errorQ.first()) par
  outQ.enq(pos.read(), inQ.first);
  inQ.deq();
  errorQ.deq();
  pos.write(pos.read()+1);
```

Figure 5-18: HW-centric Rules Calculating the Error Magnitude if blocks in the Reed-Solomon Pipeline. Forney's algorithm has been decomposed into functions f0-ff, each corresponding to one iteration of the original loop.

Figures 5-19 and 5-20 depict the hardware microarchitecture inferred for the SW and

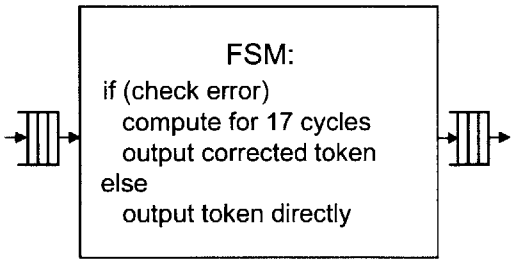


Figure 5-19: hardware microarchitecture inferred from BCL in Figure 5-17

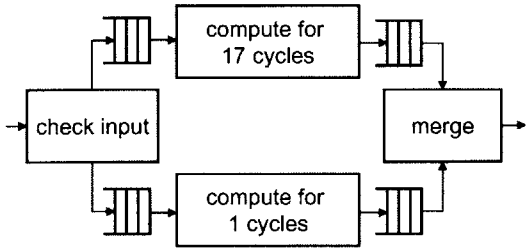


Figure 5-20: hardware microarchitecture inferred from BCL in Figure 5-18

HW-centric BCL implementations of Forney's Algorithm. The differences between these two are substantial. The task of inferring an efficient software implementation from this relatively elaborate micro-architectural description is much more difficult. Not only do we need to infer a looping structure from the pipeline implementing Forney's algorithm (as we did with the IFFT example), but we also need to infer the outer loop with the conditional body. Inferring this structure would give us a perfect schedule, but even with the perfect schedule, the challenges of compiling the generated C++ efficiently are identical to those discussed in the context of the IFFT example. A similar slow-down is observed between the alternate implementations.

5.4.3 The Ultimate Implementation Challenge

In the quest for true portability of BCL programs between HW and SW, one of two things must occur: either we should be able to infer efficient HW microarchitectures from SW-centric rules, or we must infer efficient SW from rules describing a more detailed HW microarchitecture. Though the BCL language does make certain aspects of the problem easier (explicit resources, lack of memory aliasing, etc.), we argue that inferring efficient microarchitectures from SW-centric rules is the same problem which many C-to-gates technologies [] attempt to address. In the general case, this problem was deemed by Agarwal *et al.* to be infeasible [7].

The result of this conclusion is that the only chance we have of achieving portability using BCL is to specify HW-centric designs and rely on more sophisticated techniques to infer efficient software implementations. For the subset of those designs where the compiler

can infer an efficient schedule, we may *approach* the ideal software performance. Unless the compiler can infer the *actual* program flow control (looping, conditional dependencies, etc.), we will never be able to actually achieve native software performance. As these examples show, compiling each rule to efficient in software is an important first step, but more aggressive transformations are required. Once a perfect schedule has been generated which allows us to execute rules unconditionally (if such a schedule even exists), we need to combine these sequences by merging them into single code blocks which allow us to eliminate many intermediate storage steps. Chapter 6 proposes a technique for determining the perfect schedule; the task of rule merging and buffer elimination is left as future work.

5.5 Performance of Generated Code

In Chapter 2, we considered the suitability of a set of applications to acceleration through the use of specialized hardware. In this section, we examine a group of BCL programs compiled entirely to software to evaluate the effectiveness of the BCL SW compiler. There are three ways in which we can improve upon a direct translation of the BCL Rules. The first, as we discussed at some length in Section 5.3, are transformations which can be performed on the rule body to improve the quality of the generated C++ methods which implement the rule. The second is the choice of a rule execution strategy or schedule, and the third is rule merging and the removal of intermediate storage elements. Chapter 6 proposes a technique for the selection of the schedule which is a precondition for removing intermediate storage. In this section, we will examine the impact of optimizing the implementation of the rule bodies and of improving the rule scheduling strategies.

The benchmarks represent typical BCL kernels, some of which were taken from the applications in Chapter 2, and all of which are representative of common hardware idioms. Pursuant to the discussion in Section 5.4, these benchmarks are written in such a way that they compile to efficient FPGA implementations. IFFT and IDCT were taken directly from the Vorbis and JPEG benchmarks. Diamond and Double-Diamond are pipelined structures with varying degrees of concurrency taken from the Reed-Solomon benchmark. Diamond consists of two independent synchronous pipelines which originate at a single source node

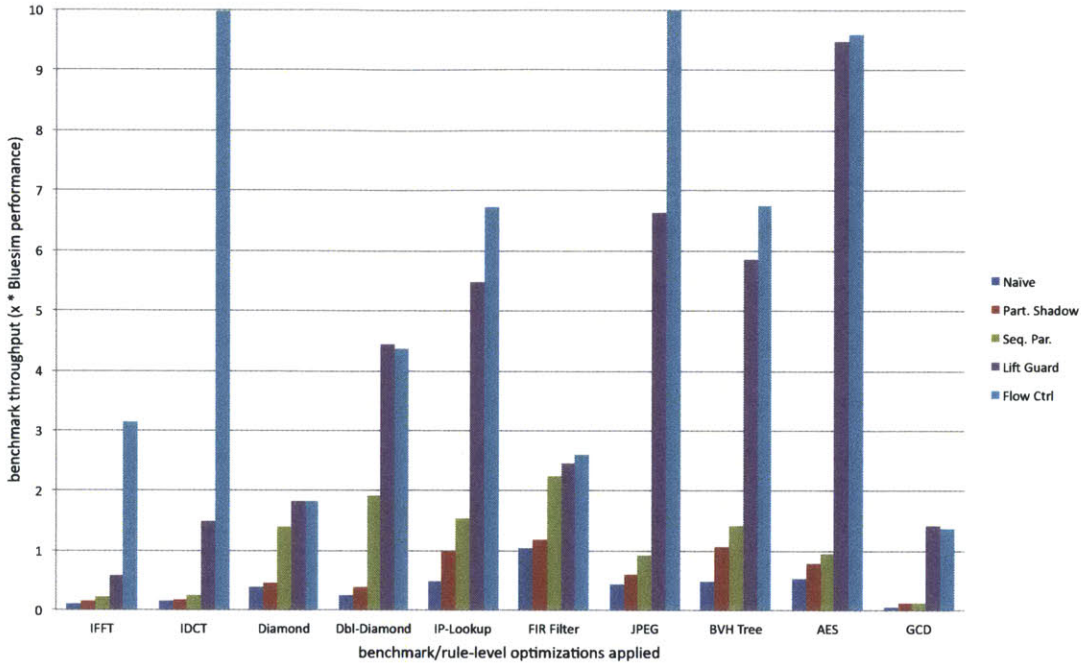


Figure 5-21: Effects of Rule Transformations on Benchmark Performance

(fork) and terminate at a common sink node (join) . Double-diamond consists of a recursively generated instance of Diamond in which the increased concurrency produces a much larger choice of valid rule schedules. IP-Lookup and FIR Filter are both common Bluespec examples. BVH Tree traversal from the Ray Tracing benchmark was chosen due to its inherent (unavoidable) dynamism, and AES and GCD are straightforward finite state machines with differing numbers of rules and varying complexity.

The impact of a particular optimization will vary from application to application. Within the context of a single application, optimizations will also vary in their degree of usefulness depending on how the rules are scheduled. In the context of single threaded software (the only one we consider in any detail), schedules are differentiated primarily by the number of guard failures they encounter, and optimizations which lower the cost of a guard failure will clearly have a greater impact when more rules fail. Since rule failures are often unavoidable, due either to our inability to ascertain the perfect schedule, or to the inherent dynamism in the program's dataflow, is may be useful to consider a range of schedule qualities when evaluating the effects of rule optimizations.

Figure 5-21 shows the incremental contributions of the rule-level optimizations discussed in Section 5.3 in programs executed under the reference (round-robin) schedule. Columns in the graph are grouped by benchmark: the left-most column in each group shows the performance of a naive rule translation using the reference scheduler to drive the program execution. As we move right through the group, each column shows the change in performance by adding one additional optimization. The right-most column in each application shows the cumulative effects of all the optimizations. We use Bluesim from Bluespec Inc. as the performance baseline, so all results are *normalized* to the performance of the baseline implementation. It should be noted that the order in which these optimizations have been added is not arbitrary. For example flow-control optimizations less effective without method in-lining.

Excluding JPEG, we see an average of 6X speedup over Bluesim, which we attribute to more aggressive flow-control optimizations inside the rules, and the fact that the BCL-generated SW does not implement the Esposito schedule. This suggests that even the simple round-robin reference scheduler gives us some advantage. Another observation is that our performance advantage depends greatly on the benchmark. In general, as the number of rules in the benchmark increases so too does the impact of the optimizations. The results in Figure 5-21 show that the rule-level transformations are *sine qua non* to efficient software. Because of this, it is a bit silly to consider them in isolation since the BCL compiler will never *not* apply any of them. The next question to ask is: "How much could we further improve the performance of these benchmarks if we put more effort into the scheduling of the rules?"

In Figure 5-22, we show the improvement in performance when executing the fully transformed rules under a so-called "perfect" schedule over their execution under the reference schedule (labeled *flow-control* to correspond to the fully-optimized series in Figure 5-22). Once again, these numbers are normalized to the baseline performance of Bluesim. Under the perfect schedule, we see an average of 20X performance improvement. By our definition, a perfect schedule is one which minimizes the number of dynamic guard failures encountered in the execution of the program. Programs whose underlying dataflow is fully synchronous can theoretically be executed with no guard failures. The presence

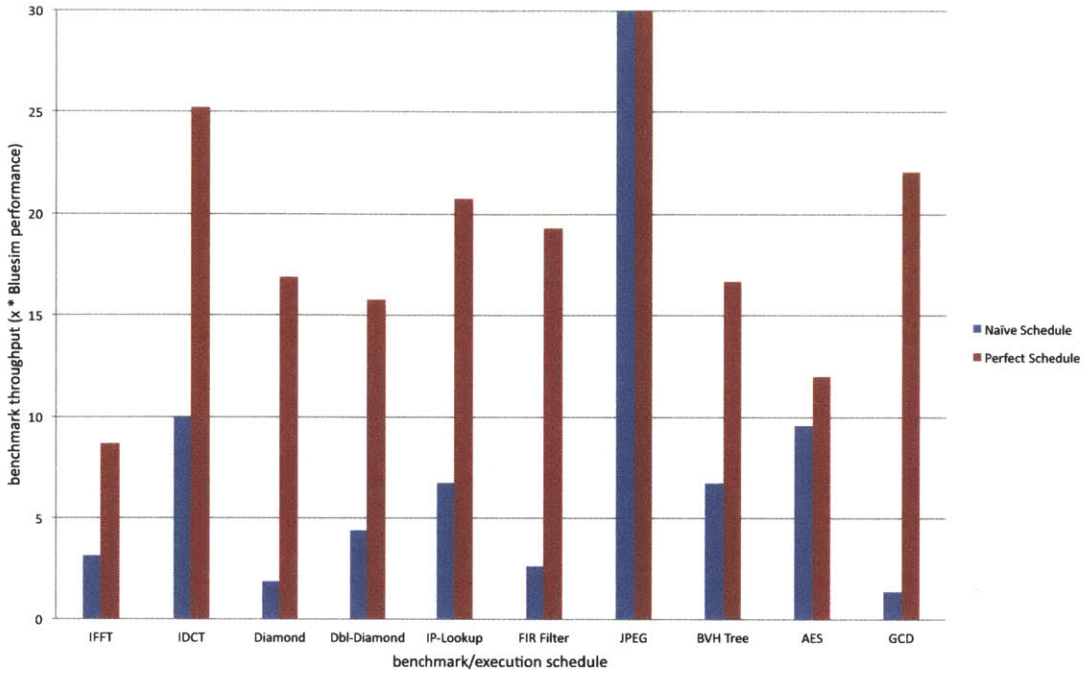


Figure 5-22: Performance Impact of a "Perfect" Scheduling Strategy which Minimizes Guard Failures

of dynamic dataflow, on the other hand, necessitates some guard failures since guards *are* the way in which this dynamism is expressed. What is interesting to note is how close to "ideal" performance we can get by concentrating solely on the quality of the rule body in certain applications. For programs with small number of rules such as AES or FIR, we can achieve up to 85% of ideal performance without any scheduling improvements. As the number of rules increases in JPEG, BVH, or IP-Lookup, the chances of the reference scheduler selecting a disabled rule increases substantially and relative performance under the reference scheduler degrades accordingly.

Finding the perfect schedule is difficult or sometimes impossible, so it is worth while considering other scheduling strategies which require less effort to compute. In Figure 5-23, we compare the performance of each benchmark executed using its perfect schedule to the performance under two other scheduling strategies which require much less expensive program analysis. The performance numbers in Figure 5-23 are given as a fraction of the of the throughput achieved by the perfect schedule, and all schedules are applied in combination with all the rule-level transformations discussed previously. The series are

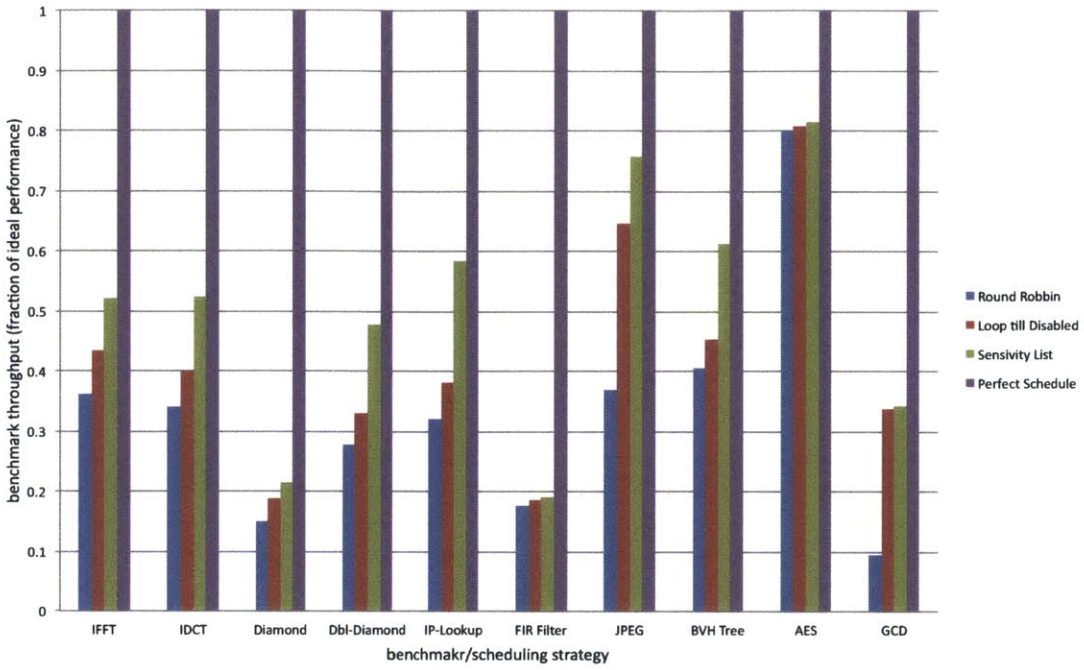


Figure 5-23: Performance Impact of Dynamic Scheduling Strategies

labeled with the scheduling strategies as follows:

- **Round-Robin:** round-robin approach employed the reference schedule
- **Loop till Disabled:** execute each rule until it is no longer enabled
- **Sensitivity List:** re-attempt when certain conditions are met
- **Perfect Schedule:** minimize the number of guard failures

The Loop till Disabled strategy employs the same order as the reference schedule, but executes each rule until it disables itself. This is a clear aberration from the fairness in the reference schedule, which can be problematic if a design is under-specified. The Sensitivity List strategy involves the static computation of the set of rules in the system which can enable every rule and encodes these sets in the scheduler directly. Using this information, we only attempt to re-execute a rule which has failed when one of it's enabling set successfully executes. The throughput of applications whose cost-of-rule-failure is high may improve under this strategy. If the cost-of-rule-failure is too low, the additional dynamic record keeping of the sensitivity list will nullify the reduction in failed rule attempts. Some

of the applications with small numbers of rules (AES, GCD, FIR) achieve relatively high throughput under the approximate scheduling strategies, but as the number of rules increase in the larger examples, we can see that the performance penalty of an imperfect scheduling strategy increases significantly. We can conclude from the comparisons of the different schedules that there is great value to getting as close as we can to the "perfect" schedule. The discussion of how is the subject of the next chapter.

The performance of the benchmarks under the perfect schedule could be further improved upon with rule merging and buffer elimination, but since we have not explored the automation of this step, we do not include it in our results. Alternately, we could have used hand-written (targeted for efficient software) implementations of each benchmark to measure the ideal performance. Though useful, this comparison is somewhat arbitrary (as discussed in Section 5.4) since we are comparing two completely different algorithms. Nonetheless it deserves some attention.

The final point of comparison is to see how much the performance of the benchmarks would improve if we were to re-implement then in "SW-style" as previously discussed. We re-implemented IFFT, IDCT, JPEG, and AES. The "HW-style" implementations achieved between 60% to 80% of their "SW-style" counterparts. We suspect that much of the performance disadvantage of the "HW-style" implementations could be removed with the aggressive rule-merging and buffer elimination transformations enabled by the implementation of the failure-minimizing or perfect schedules. It is important to remember that the change between HW and SW styles in an application requires a fundamental change in the code structure, something which we feel is impossible to automate in the general case [7].

The take-away from this discussion on SW generation is simple. As with every other programming language, performance ultimately depends on *how* the program is written. We have demonstrated BCL programming styles which enable the straightforward generation of efficient hardware, and others which enable the generation of efficient software. We have shown that efficient software can sometimes be generated from a program written in the "HW-style", but that going from "SW-style" BCL to efficient hardware is difficult. In this sense, we must start with the assumption that the programmer knows what he is doing and because of this has a good idea of the ultimate program partitioning. The compilation

techniques described here and in Chapter 6 can then be applied to the code at the interfaces.

5.6 HW Generation

Techniques for hardware generation from rules are well understood and used successfully in the commercially available compiler (bsc) from Bluespec Inc. With the exception of loops and sequential composition, BCL can be translated directly to legal Bluespec SystemVerilog (BSV), which is then compiled to Verilog using bsc. We give a short sketch of the strategy employed by bsc, and leave the reader to follow the references for further details.

To implement a rule directly in hardware, we require some notion of shadow state so that we can unwind a computation if we encounter a guard failure in the evaluation. These shadows can require a substantial hardware cost if they are implemented via stateful constructs. However, if we store them ephemerally in wires, they become cheap. The guard drives a multiplexer on the registers holding the module state, which updates them only if the guard evaluates to true. In synchronous hardware, we can guarantee that all shadows are implementable in wires as long as each rule is executed in a single clock cycle. This is the primary optimization which enables the compilation of guarded atomic actions into efficient synchronous circuits [34, 39, 40].

Since, loops with dynamic bounds can't be executed in a single cycle, such loops are not directly supported in BSV. A technique has been proposed to introduce rules which can be executed over multiple cycles [43, 53] which provides a solution to this limitation.

Chapter 6

Rule Scheduling in Software

Bluespec programs consist of explicitly specified state, and a collection of guarded atomic actions or rules. Rules are functions of type $state \rightarrow state$ which are executed atomically. If a guard failure is encountered during the evaluation of a rule, no state is updated. The non-deterministic execution semantics of Bluespec are specified in Figure 4-2

The faithful implementation of rule atomicity and the generation of an effective rule schedule are the primary challenges in producing efficient software. Chapter 5 presented strategies for efficiently enforcing rule atomicity involving both static analysis and run-time support. To address the challenge of generating effective rule schedules, we also propose both static and dynamic techniques. In this chapter, we present some simple dynamic strategies to scheduling rules in software, and then discuss at length a novel scheduling approach based on the abstract evaluation of rules using an SMT solver.

Any Bluespec compiler must implement a scheduling strategy which is faithful to the semantics specified in Figure 4-2. Of course, there are many *legal* strategies, such as always choosing the same rule, which are easy to implement efficiently but not particularly useful. We will ignore these "theoretically interesting, but practically useless" cases, and spend our time examining strategies of a more practical bent.

Definition 1 (Enabled/Disabled). Rule r is said to be *enabled* in state s if the evaluation of r in s does not result in a guard failure. Conversely, if guard failures are encountered, r is said to be *disabled* in s. As shorthand, we employ the predicate function $en(r, s)$

throughout the remainder of this chapter, which evaluates to true if and only if r is enabled in s .

6.1 A Cost Model for Executing Rules

Following the transformations described in Chapter 5, rules will be left in one of two forms. Group **A** consists of rules in which all the guards have been completely lifted, while the guards of rules in group **B** have not. The presence of sequentially composed actions within a rule is the only thing which prevents complete guard lifting. For rules in **A**, the complete lifting lets us evaluate all the guards before computing the state updates. Rules in **B** retain the possibility of a guard failure *after* some state updates have been applied. Since rules in **B** necessarily contain actions composed in sequence, the BCL type system can guarantee that a hardware compiler will never deal with this case. This will change when the hardware back-end (BSC) supports sequential composition of actions within rules. In Software, if all actions within a rule are composed in parallel and all sub-module interface methods are decomposed into their guard and body components, the rule will fall into group **A**. With all rules falling into either group **A** or **B** we can begin to reason about the cost of executing a rule in software. Figure 6-1 outlines a crude estimation of the cost of executing the C++ code corresponding to rules in both categories.

	Guard Evaluation	Body Evaluation	Commit Updates	Failure	Success
Group A	1	1	0	1	2
Group B	0	2	1	2	3

Figure 6-1: A Simple Cost Model for Rule Execution

The execution of a rule in software can be broken down into four components. There is the code associated with guard evaluation, code associated with computing the state updates, code which implements shadow state, and code which implements roll-back. Since speculative rules will use either shadow or rollback (not both), a rule can have code belonging to at most three of these four components. We assign equal computational weights to each of these four tasks, and while there is clearly a lot of variation between the code

implement the guard expressions and the code implementing the calculation of the state updates from rule to rule, this crude estimation will serve as a starting point.

The weights listed in Figure 6-1 for group **A** are relatively self explanatory: if the guard evaluates to **true**, we can execute the rule body and apply the state updates *in situ*. In the case that the rule is enabled, there is no overhead associated with its execution. If the rule is disabled, the failure overhead of 1 represents the wasted work of evaluating the guard.

The weights listed in Figure 6-1 for group **B** are slightly more complicated. Since we are not able to lift the guards completely, their evaluation is tantamount to executing the rule itself, so the estimated weight of the rule guards is added to that of the body. Failure to lift guards completely implies there the presence of a sequential composition with a dependency between an update performed inside the rule and the rule's guard. This means that the enforcement of atomicity requires the introduction of shadow state or a roll/back mechanism. A cost of 2 in the failure case and 3 to success implies the use of statically allocated shadow state, since a successful execution requires additional work to commit the shadowed updates while the updates to the shadow state by a failed attempt can simply be ignored. Had we chose to use a roll-back mechanism instead of shadow state, the weights would reversed. Once again, an arbitrary cost of 1 has been assigned to this overhead. We can now use this cost model to compare the relative merits of scheduling strategies.

6.1.1 Comparing Schedules using the Cost Model

Even with effective guard lifting and an efficient run-time designed to lower the cost of rule failures, finding a rule which will not fail invariably increases the work performed by a dynamic scheduler. This implies an unavoidable overhead for the dynamic execution of rules in BCL, a phenomenon which has been well documented [].

As described in Chapter 5, we generate a single C++ method corresponding to each rule which executes the body of that rule. This method returns an error code indicating whether or not guard failures were encountered during the rule evaluation. In the case where guards have been completely lifted, C++ code is a simple `if` statement. When guards remain in the rule body due to the presence of sequential action composition or I/O, more complicated

```
module mkSimpleExample = do

 IStream#(Int) i <- mkIStream;
 OStream#(Int) o <- mkOStream;
 FIFO#(Int) f1 <- mkSizedFifo(1);
 FIFO#(Int) g1 <- mkSizedFifo(1);

 rule a1 = do
   x <- i.next
   when (x >= 0) f1.enq(x)

 rule a2 = do
   x <- i.next
   when (x < 0) g1.enq(x)

 rule b1 = do
   o.next(f1.first)
   f1.deq

 rule b2 = do
   o.next(g1.first)
   g1.deq

endmodule
```

Figure 6-2: A BCL program with dynamic dataflow and non-deterministic choice

code is required to shadow the updates and commit them upon successful rule completion. The high-level rule driver is agnostic to the rule internals, and simply invokes the methods with the assumption that atomicity is correctly maintained. A simple program is presented in Figure 6-2 for the sake of demonstrating how to compute the overall cost of a generated rule schedule.

The BCL program in Figure 6-2 specified a structure with two pipelines (rules [a1,b1], and [a2,b2]). A dynamic dispatch mechanism consisting of rules a1 and a2 enqueues tokens into one pipeline or the other. For argument's sake, we will assume that input streams can be modeled by an infinite stream of values, and that output streams can be modeled by an unbounded buffer. In BCL terms, this means that the "canGet" method on the input stream and the "canPut" method on the output stream always return true. Later on, we will

show how to relax this condition. The first schedule we will consider is a simple round-robin dynamic scheduler. In our first scheduling attempt, we assume no knowledge of the underlying dataflow structure of the program and simply sort the rules in lexicographic order by name, invoking them in a while loop as shown below:

```
while(true){
  a1();
  a2();
  b1();
  b2();
}
```

With this schedule, less than half of all rule invocations will complete successfully. As the rule bodies grow in complexity, the overhead of rule failures begins to dominate execution time. By the cost metrics given in Figure 6-1, this rule schedule requires 6 units of work to write each token to the output stream. If we changed the rule body implementations such that their C++ implementations fell into group **B** instead of **A**, the cost would increase to 10 units of work.

In the next schedule, we will attempt something slightly more intelligent. Assuming the equal likelihood that an input token is greater than or less than zero, the schedule listed in below could write an token to the output stream with an average of 4.5 units of work (for each input there is a 50% chance the first test will fail):

```
while(true){
  if(a1())
    b1();
  if(a2())
    b2();
}
```

If rule implementations were all in group **B**, the above schedule would generate one output token with an average of 5.5 units of work. In the worst case (all rules in group **B**), the second schedule reduces the work by 50%, while in the best case (all rules in group **A**), we see a 20% improvement. Given only the unmodified methods implementing each rule and without any more information about the distribution of input tokens, the second schedule

is optimal. Our goal is to generate it automatically.

In this trivial example, the performance improvements between an arbitrary schedule and an optimal one are modest, but for larger designs the improvement will be greater. This example is useful because it contains the two elements which make scheduling rules in Bluespec programs a challenge: non-deterministic choice and dynamic dataflow. Non-deterministic choice the result of reachable states where more than one rule is enabled in which the compiler must choose which rule to execute.

6.2 Dynamic Rule Scheduling Strategies

For those Bluespec programs whose underlying dataflow is synchronous [45], it is always possible to encode a branch-free schedule directly and avoid the overhead of dynamic execution all-together, though finding this schedule may be an expensive proposition. For those programs with dynamic behavior of if we simply don't want to exert the effort of finding the synchronous schedule, we can still generate more intelligent rule schedulers which are able to choose the next rule to execute more efficiently.

Statically computing inter-rule dependencies for construction of a sensitivity-list is one such approach, where the successful execution of a particular rule can provably enable or disable a particular subset of rules. While this doesn't remove the possibility of choosing the wrong rule, it does reduce the number of rules the scheduler needs to choose from at any given point in time. A naive (though surprisingly effective) extension of this strategy which doesn't require any compiler analysis is to execute a rule until it disables itself.

Additional strategies could involve predicting the next rule based on past behavior in a manner similar to branch prediction, though this strategy has not yet been implemented. The challenge with dynamic techniques is to keep their cost below that of evaluating the actual rule guards.

6.2.1 Parallel Rule Execution in Software

One major advantage of using Bluespec to describe hardware is the relatively low overhead of an effective parallel implementation strategy [34]. The low overheads of parallel hard-

ware are due primarily to the parallel nature of the targeted execution substrate (FPGA, ASIC), and allows a hardware implementation to take advantage of the fine-grained parallelism by executing many rules in parallel. Generating parallel software, on the other hand, presents a far greater overhead due to the inherently sequential nature of the underlying execution substrate (CPU). If we attempted to implement the Esposito schedule directly in software, the overhead of spawning a thread to execute each rule would be massive. Instead, our approach to multi-threaded software is to require the programmer to specify multiple software partitions with exactly the same partitioning mechanism used to specify hardware partitions. Each SW partition is assigned its own thread, and lightweight IPC mechanisms can then be used to implement the interface FIFOs between the software domains.

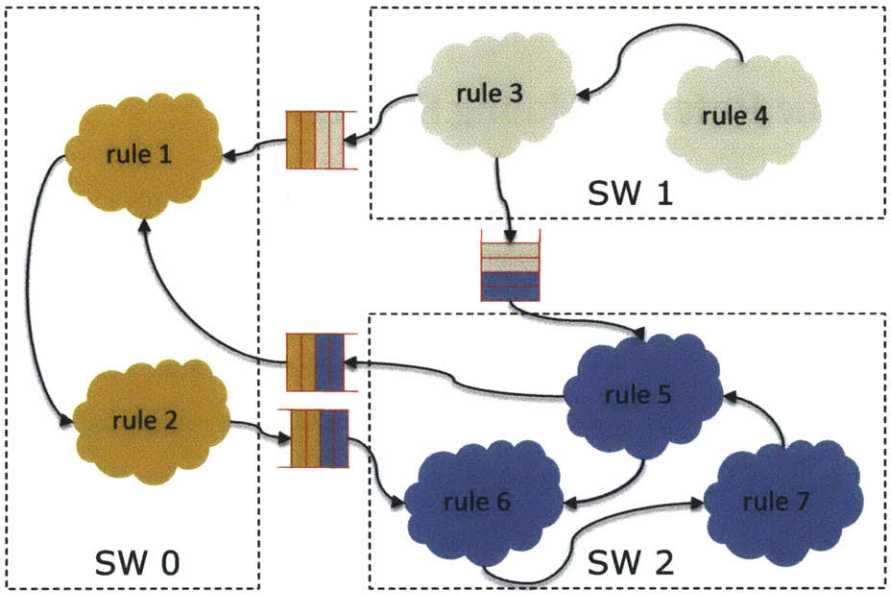


Figure 6-3: Generating Multi-threaded Software using BCL

In Figure 6-3, we see a program which has been partitioned up into three computational domains: SW0, SW1, and SW2. Instead of creating a single software partition out of these three domains, the compiler can choose to create three separate software domains. Because of the safety guarantees enforced by the BCL type system among computational domains, the assignment of a separate thread to each domain requires no further analysis. Since the operating system is responsible for scheduling the execution of concurrent threads, the problem of scheduling Bluespec rules in software can be restricted to the serial execution

of rules in a single software domain. The concurrency achieved between multiple software domain will depend on may of the same factors which affect HW/SW concurrency, such as communication granularity and synchronization points. The overall speedup of such an implementation depends on the equal division of labor among the threads and a low compute to synchronization ratio.

Finally, the automatic parallelization of the execution of a BCL program using a pool of worker threads and a work list of rules is another viable solution. For this to work, care needs to be taken to ensure that the sequential consistency is maintained. Since concurrently executed rules might not be conflict free, additional support for transactional memory could provide sufficient guarantees. Whether the computational weights of the rules can justify this overhead has yet to be investigated.

6.3 Static Rule Scheduling Strategies

For states in which we can guarantee that a rule will not fail, not only can we execute it *in situ* but we can avoid evaluating the guard all-together. For example, if we know that rule a *always* enables b, a successful execution of a can always be followed by executing the state updates b without evaluating b's guards. If we can further determine that always executing a and b in this manner will not introduce deadlocks, we can reduce the nondeterminism in our system by reducing the number of rules which need to be considered for execution by the scheduler. The remainder of this chapter examines approaches to determining when this style of rule-merging is safe. In cases of synchronous dataflow, the overhead associated with nondeterminism in the spec can be eliminated entirely.

The scheduling of dataflow graphs is a subject of much research [], and at its core each BCL program is describing some kind of dataflow graph. Consider the Bluespec program shown in Figure 6-4. The annotated dataflow graph underlying this program is shown in Figure 6-5. If we could easily infer the structure of the dataflow graph underlying each BCL program, then we could simply leverage known scheduling techniques for its efficient execution. For this example it is easy to infer the dataflow graph, and once complete, a simple analysis of the rules revealed its synchronous nature. In general, though, the

```
module mkSimpleExample = do

 i <- mkReg(0)
 o <- mkReg(0)
 x <- mkFIFO(2)
 y <- mkFIFO(2)
 z <- mkFIFO(2)
 w <- mkFIFO(2)

 rule a when (True) seq
   i.write(i.read()+1);
   x.enq(i.read());
   y.enq(i.read());

 rule b when (True) seq
   t0 <- x.deq();
   t1 <- x.deq();
   z.enq(t1);
   z.enq(t0);

 rule c when (True) seq
   t0 <- y.deq();
   w.enq(t0);

 rule d when (True) seq
   t0 <- z.deq();
   t1 <- w.deq();
   t2 <- z.deq();
   t3 <- w.deq();
   o.write(o.read+f(t0,t1,t2,t3));

endmodule
```

Figure 6-4: A BCL program with synchronous dataflow and a simple state-abstraction function

construction of this graph is difficult. A Naive translation would assign a node to each rule, but determining the edges and the rates at which tokens are produced and consumed is prohibitively difficult. Since the Bluespec language is much more general than SDF, in many cases the rates cannot be statically computed at all. Even though more general dataflow models such as Token Flow (which would enable certain types of analysis) are

computationally equivalent, Bluespec's use of registers, register-files, memory structures, etc. make the translation task impractical.

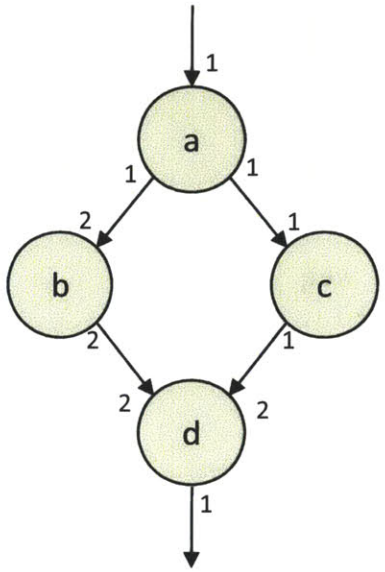


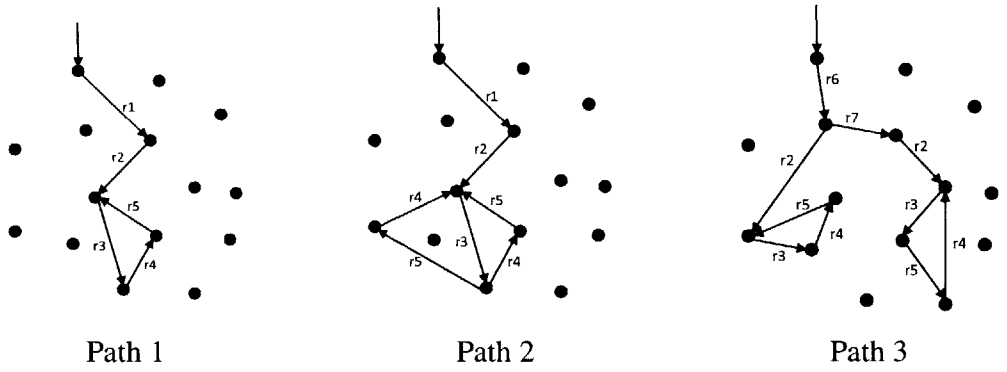
Figure 6-5: Synchronous dataflow graph generated by the BCL program in Figure 6-4

6.3.1 Reasoning about Explicit State

Since translating the Bluespec program to a more formal dataflow formalism is impractical, one possible solution might be to reason about the state directly. A Bluespec program is simply a state transition system, and in the corresponding state transition graph, there is an edge from node a to node b labeled r iff rule r is enabled in a and produces state b after applying the state updates. Since Bluespec programs are nondeterministic, a node in the state transition graph may have an out-degree greater than one. It follows naturally that if there is a cycle in the state transition graph, the rule sequence corresponding to the labels of the edges on that path represents a valid periodic schedule which can be executed repeatedly without experiencing any guard failures.

Before delving into how to detect cycles, let's look at three concrete paths through a state transition graph, shown in Figure 6-6. In Path 1, we can see that the cycle represented by rule sequence `([r3,r4,r5])*` is reachable from the initial state by executing the

sequence `[r1, r2]`. Path 2 shows alternate cycles, implying that after executing `r3`, the sequence `[r4, r5]` is equivalent to executing the sequence `[r5, r4]`; this type of symmetry is common in Bluespec programs. Path 3 shows an example of non-confluent, or divergent behavior. Either cycle in Path 3 is a valid schedule, since the execution semantics of Bluespec do not contain any notion of fairness. As you might imagine, this is a weakness of any approach based on reasoning about explicit state, since programmers tend to rely on fairness, even if no formal guarantees are made.



Path 1 Path 2 Path 3

Figure 6-6: Paths through a State Transition Graph

6.3.2 A Direct Solution Using Sketch

The explicit search in a finite state transition graph is something which can be encoded as a sequence of satisfiability problems. As shown in Figure 6-7, this search problem can be expressed directly using Sketch [56] where we instruct the tool to identify a reachable cycle in the state transition graph.

Asking the query is surprisingly simple: The first line declares the initial state of the system specified by the BCL program. The second line instructs sketch to find a rule-sequence of length zero or greater, which can be thought of as the schedule "prefix" (this sequence could correspond to an initialization loop, for example). **repeat** is a Sketch primitive whose first argument indicates the number of times the second argument (a statement) should be invoked. The hole (??) in the first argument leaves the length of this prefix unspecified. Shown in Figure 6-8, selectValidRule is a generator whose in-lining semantics

```
harness void main(){
  State state = newState();
  repeat(??) selectValidRule(state);
  State snapshot = copyState(state);
  selectValidRule(state);
  repeat(??) selectValidRule(state);
  assert compareState(state,snapshot);
}
```

Figure 6-7: Sketch Harness for Synchronous Dataflow Search

permit it to select a different rule each time it is invoked. With each invocation from within a repeat statement, selectValidRule can return an instantiation of it's body with different values assigned to the holes. The result is that a different rule may be invoked at each inlined invocation. Next, we take a snapshot of the state which corresponds to the entry point of the cycle we are searching for. Using the same mechanism with which we specified the schedule prefix, we then instruct Sketch to find a rule sequence of one or more rules which will put us back in the state produced by the prefix. We give the Sketch solver an upper bound unrolling the **repeat** command. Once Sketch has found a solution and elaborated the main harness, this code can be used directly to drive the execution of the BCL program.

```
generator void f(State state){
  if(??)
    a(state);
  else
    if(??)
      b(state);
    else
      if(??)
        c(state);
      else
        d(state);
}
```

Figure 6-8: selectValidRule implementation for the BCL program in Figure 6-4

Procedures used to translate Bluespec rules, actions, and expressions to Sketch, are

shown in Figures 6-10, 6-9, and 6-11, respectively. The helper functions used in action translation given in Figure 5-6, and the Module translation procedure given in Figure 5-7 can be used directly. The translation from BCL to Sketch is relatively straightforward because assertions in Sketch directly capture the semantics of guard failures. The only additional utility methods required are a copy constructor and a comparison procedure used to assert the equality of two state elements. The copy constructor is used to take state snap-shots, compareState simply recurses the module hierarchy and calls compareState on each sub-module, and a straightforward implementation of compareState at primitive leaf modules simply equates the objects memory in a bit-wise manner.

```
TA :: Env × BCL-Action → CStmt

TA ρ [[ r := e ]] =   se; ρ[r].write(ce);
          where   (se, ce) =   TE ρ [[ e ]]

TA ρ [[ if e then a ]] =   se; if(ce) {TA ρ [[ a ]]}
              where   (se, ce) =   TE ρ [[ e ]]

TA ρ [[ a1 | a2 ]] = cs1; cs2; (TA ρ1 [[ a1 ]]); (TA ρ2 [[ a2 ]]); pm; ms;
        where  (cs1, ρ1) =   makeShadow ρ
               (cs2, ρ2) =   makeShadow ρ
               (pm, ρ3) =   unifyParShadows ρ1 ρ2
                    ms =   commitShadow ρ  ρ3

TA ρ [[ a1;a2 ]] = (TA ρ [[ a1 ]]); (TA ρ [[ a2 ]]);

TA ρ [[  a when e  ]] = se;  assert(ce); (TA ρ [[ a ]])
            where  (se, ce) =   TE ρ [[ e ]]

TA ρ [[  t = e in a  ]] = se; t=ce;  (TA ρ [[ a ]])
            where  (se, ce)   =   TE ρ [[ e ]]

TA ρ [[ m.g(e) ]] =   se; (ρ[m].g(ce));
          where   (se, ce) =   TE ρ [[ e ]]

TA ρ [[ loop e a ]] =    while(true){se;if(!ce){break;} (TA ρ [[ a ]])}
            where   (se, ce) =   TE ρ [[ e ]]

TA ρ [[ loopGuard e a ]] =   try{while(true){se;  if(!ce){break;}ca;}}catch {}
                  where   (se, ce) =   TE ρ [[ e ]]
                               ca =   TA ρ [[ a ]];
```

Figure 6-9: Translating BCL Actions to Sketch

genRule ρ [[(Rule name a)]] = `void name(){`**TA** ρ [[a]]`}`

genAMeth ρ [[(AMeth name v a)]] = `void name(t v){`**TA** ρ [[a]]`}`

genVMeth ρ [[(VMeth name v e)]] = `rt name(t v){se; return ce;}`
where (se,ce) = **TE** ρ [[e]]

Figure 6-10: Translating BCL Rules and Methods to Sketch

The state-space described by the BCL program in Figure 6-4 is large, so if the Sketch program has any hope of finding a cycle of reasonable length, a state abstraction function becomes essential. By naively defining the compareState function to assert equality of all state variables, we are essentially asking Sketch to search over the full state transition graph and even in this small program it has no chance of finding a sequence of reasonable length, since 'i' and 'o' (both 32-bit integers) would have to wrap around. If, on the other hand, we recognize that the actual value of 'i' and 'o' in the example in Figure 6-4 have no effect on which rules can fire, then we can exclude them from compareState, resulting in the following implementation:

```
bit compareState(State s1, State s2){
    return (compareFIFO(s1.x, s2.x) && ... &&
            compareFIFO(s1.w1,s2.w1) );
}
```

The function compareFIFO takes two FIFOs as arguments returns true iff the contents of the two FIFOs match one-to-one. With this level abstraction, Sketch converges quickly to a schedule of length 4: `[a,b,c,d]*`. Being even more aggressive, we could encode the fact that in the context of this program, the value of the tokens in the FIFOs can never cause a guard to fail, since all guards are dependent *only* on the number of tokens in the FIFOs. Encoding this fact further improves the compression of the state transition graph and makes the search even more efficient.

The difficulty with automating our use of Sketch lies in the generation of the state abstraction. Each Bluespec program requires a slightly different function, and in most cases, the distinction between control and data-path logic is not so clear. The example in Figure 6-12 demonstrates the worst case, in which an effective abstraction requires a partial

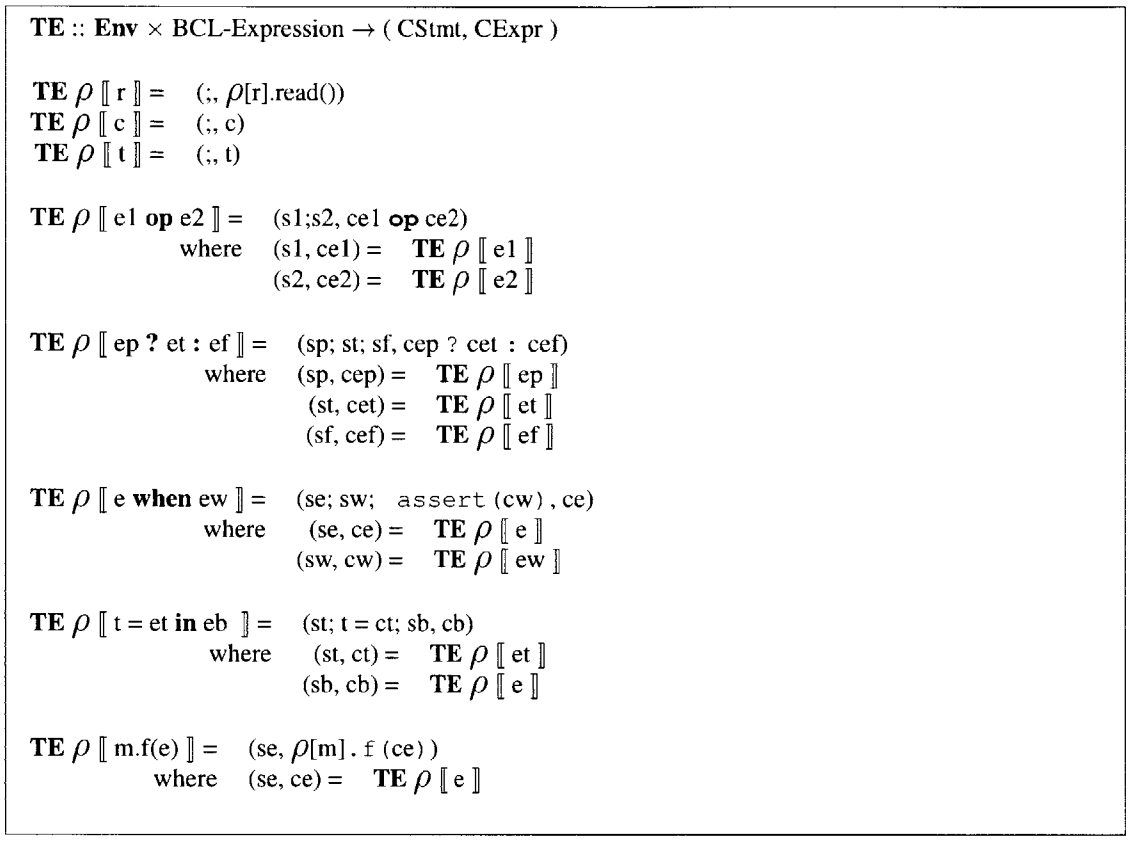


Figure 6-11: Translating BCL Expressions to Sketch

encoding of the program's behavior. One possible implementation of the compareState function for this program is as follows:

```
bit compareState(State s1, State s2){
  return ((s1.init == s2.init) &&
          (s1.init == True || s1.i == s2.i ) &&
          compareFIFO(s1.x, s2.x) && ... &&
          compareFIFO(s1.w1,s2.w1) );
}
```

Until a solution to this problem is found, the use of Sketch as part of an automated tool-chain is not practical. Additional problems with Sketch are related with its ability to scale to larger programs and the general brittleness of its implementation. Lastly the the answer returned by sketch is "all or nothing"; there is no solution to the problem if there is any dynamism in the dataflow. It would be more useful to have an approach which, if the DFG

```
module mkSimpleExample = do

 i <- mkReg(0)
 o <- mkReg(0)
 init <- mkReg(False);
 x <- mkFIFO(2)
 y <- mkFIFO(2)

 rule a when (i > 5 || init == True) seq
   init <= True;
   i <= i+1;
   x.enq(i);

 rule i when (i <= 5 && init == False)
   i <= i+1;

 rule b when (True) seq
   t1 <- x.deq();
   y.enq(t1);

 rule c when (True) seq
   t0 <- y.deq();
   o <= o+t0;

endmodule
```

Figure 6-12: A BCL program with synchronous dataflow and a difficult state abstraction function

was not fully synchronous, could at least return us some useful information which could help is incrementally improve the quality of our scheduler.

6.3.3 Encoding an FSM

The approach to analyzing BCL programs proposed in this thesis is based on the construction of an abstract state transition graph. Rather than defining a globally applicable state abstraction function, we refine the graph through local transformations. Though the graph may contain ambiguous nodes (defined as nodes with multiple out-edges with the same transition label) due either to inherent dynamism in the underlying dataflow graph or sim-

ply an abstraction function in need of further refinement, it is still useful in the construction of a more efficient software schedule.

State in Bluespec programs is finite, which means we can construct a state transition graph where nodes represent concrete states, and edges represent rule application. In this graph, each node contains one out-edge for each rule enabled in that state, and there is a path corresponding to all possible rule sequences in the program's execution. A scheduler could simply traverse this graph, resulting in a "perfect" schedule. For the example given in Figure 6-2, the state elements we need to consider are the four FIFOs and the head of the input stream; nodes in the state transition graph correspond to concrete states, or the actual values assigned to these state elements. For each concrete state, we can compute which rules are enabled, and for each rule/state pair, we can also compute the set of states generated by applying the rule updates. Where dynamic dataflow exists, the size of this set may be greater than one. For programs of even moderate complexity, the number of concrete states is very large. Not only is generating the concrete state transition graph complex, but encoding it as a rule-scheduler is likely to be more expensive than simply executing the dynamic scheduler directly at run-time.

To reduce the size of this graph, we need a straightforward abstraction function with which we can group states. If chosen wisely, this function will compress the graph while still retaining enough resolution to be useful to a dynamic scheduler. Since we are really only interested in whether or not a rule can fire in a particular state, we choose an abstraction function composed of the *enabled* predicates corresponding to each rule. Though we choose a single abstraction function to define all the nodes in our initial graph, we can refine this function locally at different points in the graph to achieve better resolution.

Definition 2 (Predicate Function Generator g). For a Bluespec program P with rules $R_P = \{R_0..R_{n-1}\}$ and state of type $state$, we define the function g_P of type $bitvector[n] \rightarrow state \rightarrow bool$ as follows:

$$\begin{aligned}\lambda\, v\ s\ \rightarrow \bigwedge_{i=0}^{n-1}\ &(((v[i] = 1)\ \wedge\ en(R_i, s))\ \vee \\ &((v[i] = 0)\ \wedge\ \neg en(R_i, s))\ \vee \\ &(v[i] = *))\end{aligned}$$

The function g_P takes as arguments a tri-state logic bit vector v of length n, and a concrete state s. The bit-vector v defines a set of concrete states by the rules which must be enabled, those which must be disabled, and those left unconstrained. The function returns a boolean value indicating the membership of s in the set characterized by v.

Definition 3 (Predicate Function p). For a predicate function generator g_P and the lexicographically ordered set v of tri-state bit-vectors of length n, the predicate function p_j of type $state \rightarrow boolean$ is defined as follows:

$$\lambda\, s \;\rightarrow g_P\; v_j\; s$$

We can define n^3 different predicate functions through the partial application of g_P. These predicate functions will be used to define abstract states as we attempt to compress the state-transition graphs and detect rule sequences.

6.3.4 Construction the Abstract State Transition Graph

Instead of concrete states, nodes in our graph will represent sets of states, or abstract states, each defined by a predicate function. Whereas a concrete state corresponds to a single assignment of state values, an abstract state in our graph corresponds to all state assignments in which the predicate function evaluates to $True$. If we are judicious about how we build the abstract state transition graph, it will be vastly smaller than its concrete counterpart. At each step in construction the graph, we will maintain the following two invariants:

1. **No Aliasing:** A a concrete state in the program being modeled can be contained in at most one abstract state represented by nodes in the graph.

2. **False Edges:** An edge in the graph will connect two nodes **iff** the transition exists in the program being modeled.

The high-level idea behind the graph refinement algorithm presented in this chapter is to begin with a course abstraction which is trivial to encode, and refine this abstraction until the graph it describes can be encoded into a more efficient schedule. This refinement process is shown in Figure 6-13. Before refinement, the graph contains a single abstract

state defined by the predicate function "$\lambda x \rightarrow True$". With a self-edge corresponding to each rule in the system, this graph permits all legal schedules, but is useless in narrowing rule choice. All rule choice in the refined graph has been removed and at each state there is a single rule (or rule sequence) which deterministically leads to the next state. This synchronous schedule is possible only because of the underlying program structure, and there are cases where some choice will remain in the fully refined graph.

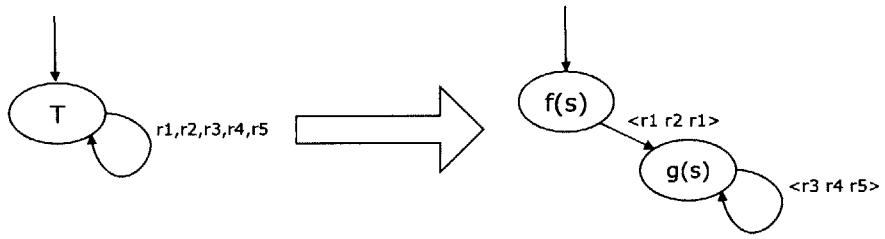


Figure 6-13: High-level Goal of Graph Refinement Procedure

The final graph will be fully resolved, meaning that there will be no abstract states corresponding to predicate encodings with high-impedance (unknown) bits. Optimizations to the graph construction involve leaving unknowns in certain nodes, but these will be considered later. Nodes in the graph are encoded by the tuple (i, v) where i is a unique node [integer] id, and v is the bit-vector encoding of the corresponding abstract state. Edges in the graph are encoded by the triplet (s, d, l), where s is the source node id, d, the destination node id, and l the name of the rule represented by transition in the graph. The graph G is encoded as the tuple (N, E) where N is a list of the nodes and E is a list of the edges. The construction of G for the Bluespec program P consisting of state S_P and rules R_P proceeds as follows:

1. $G := (N, E)$ where $N := \{(0, *^n)\}$, $E := \{(0, 0, R_i) | R_i \in R_P\}$.

   In the initial graph, a single node represents all concrete states (the predicate $True$), thus all rule transitions are self edges.

2. $(n, v) := head$ ($filter$ $(\lambda(\_, v) \rightarrow v$ $contains$ $*)$ $N)$.

   Select a node G whose abstract state encoding contains $*$. (Haskell-style pattern-matching syntax is used to bind the tuple components of n)

3. $(v_1, v_2) := (repu(v, 0), repu(v, 1))$

   Build two new abstract state encodings (v_1, and v_2) by invoking the function $repu$. This function replaces the first occurrence $*$ in n's state encoding by 0 and 1 respectively.

4. $m := length(N)$

   Assign m to the next unique node id

5. $N := (N \setminus (n, v)) + [(n, v_1), (m, v_2)]$

   Split node n into two nodes. The abstract states represented by these nodes are mutually exclusive by construction.

6. $E := filter\ valid\ (E + A + B)$, where

   $$A = map\ (replaceDest\ m)\ (inEdges\ G\ n)$$

   $$B = map\ (replaceSrc\ m)\ (outEdges\ G\ n)$$

   To maintain the graph completeness, we add paths from all of n's predecessors to m, and paths from m to all of n's successors. To maintain the "no false edge" invariant, we must filter out all edges which have been invalidated by the refined predicates. The function $valid$ is implemented using an SMT solver and returns false if the transition represented by the edge is not legal in the underlying program. In practice, we don't need to test all the edges in the graph since only those adjacent to the "split" node might possibly fail the test.

7. If no abstract state encodings in G contain $*$, return (N, E)

8. goto 2

There is the small issue of initial states, but if we keep track of these throughout the construction of the graph (which abstract states contain initial states as specified by the Bluespec source), we can prune unreachable states at each iteration. If we apply the graph construction procedure to the program presented in the previous section, we will produce the graph shown in Figure 6-14.

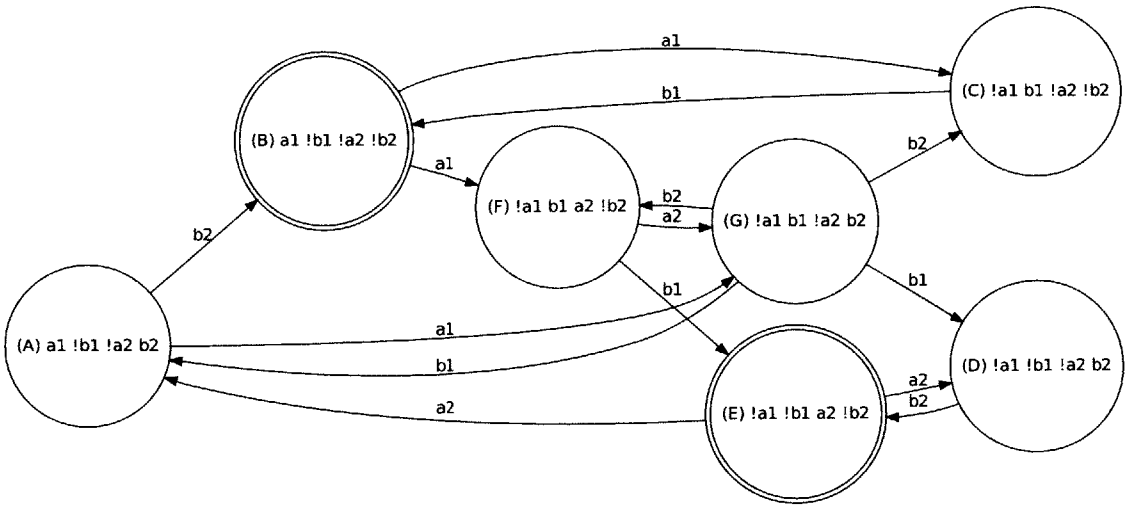


Figure 6-14: Abstract State Transition graph for example in Figure 6-2. The state vectors have been fully elaborated for readability.

The initial states of the program represented by the graph in Figure 6-14 are contained in the abstract states corresponding to nodes marked by double circle. This graph can be further simplified by removing choice between rules. Some choice is the result of dynamic dataflow and is hence unavoidable. For example, in the state B , executing the rule a1 will put us either into the state C or state F. This depends entirely on the next token on the input stream and is a reflection of the underlying dynamism in the dataflow of the program. On the other hand, for the state A, we can choose to execute either rule a1 or rule b2. This choice is a result of the nondeterminism in execution semantics of Bluespec. If we always choose b2 when in State A, we will still arrive at a schedule which adheres to the execution semantics of Bluespec. This type of pruning can be thought of as the application of a scheduling strategy, in which we always choose the rule with the highest priority from among all enabled rules at any given time. In this case, the rule prioritization is somewhat arbitrary, but further on in the discussion we will see how different strategies result in smaller graphs and more compact encodings. Applying the priority [a1,b1,a2,b2] to the graph in Figure 6-14 produces the simpler graph shown in Figure 6-15.

The Graph shown in Figure 6-15 does not directly correspond to the "ideal" schedule given previously for the program in Figure 6-2, but its direct encoding in C++ has the same

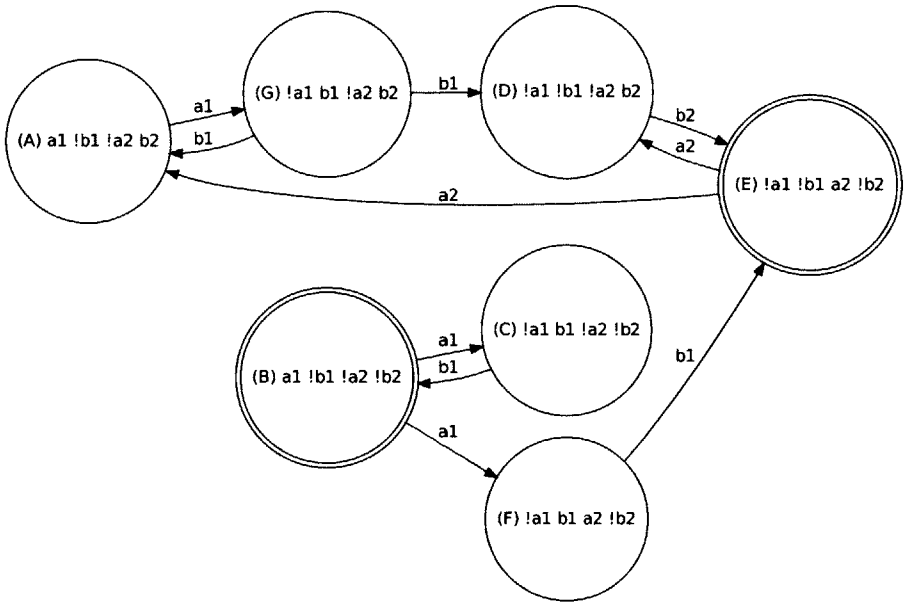


Figure 6-15: Graph from Figure 6-14 after prioritizing rules.

cost once we decompose the rule guards and bodies:

```
start:
    if (a1_guard()) {goto B;}
    goto E;
A:  a1_body();
    goto G;
B:  a1_body();
    if(a2_guard()) {goto F;}
    goto C;
C:  b1_body();
    goto B;
D:  b2_body();
    goto E;
E:  a2_body();
    if (a1_guard()) {goto A;}
    goto D;
F:  b1_body();
    goto E;
G:  b1_body();
    if (a1_guard()) {goto A;}
    goto D;
```

In this case, we were able to produce the ideal schedule using a predicate function in which the rule guards were considered at their lowest granularity. In the next section, we show an

example which requires further abstraction to achieve the perfect schedule.

6.3.5 A Slightly More Complicated Example

The example shown in Figure 6-16 contains fewer rules than the example in Figure 6-2, but is more complicated due to large number of legal schedules possible. This is due to the increased sizes of the FIFO buffers which connect the pipeline stages.

```
module mkSimpleExample = do

 IStream#(Int) i <- mkIStream;
 OStream#(Int) o <- mkOStream;
 Fifo#(Int) x <- mkSizedFifo(2);
 Fifo#(Int) y <- mkSizedFifo(2);

 rule a
   v <- i.next
   x.enq(v)

 rule b
   y.enq(x.first)
   x.deq

 rule c
   o.next(y.first)
   y.deq

endmodule
```

Figure 6-16: Deepening the Pipeline greatly increases the number of legal schedules

The abstract state transition graph for `mkLessSimpleExample` is given in Figure 6-17. This graph is substantially more complicated than the equivalent graph shown in Figure 6-14, which we constructed for `mkSimpleExample`. Once again, we will establish a rule priority, which for this example is [c,b,a]. Applying this to the graph results in the far simpler graph shown in Figure 6-18. By applying a scheduling strategy, we are no longer considering all legal schedules, thereby rendering many valid states unreachable and consequently reducing the size of the graph dramatically.

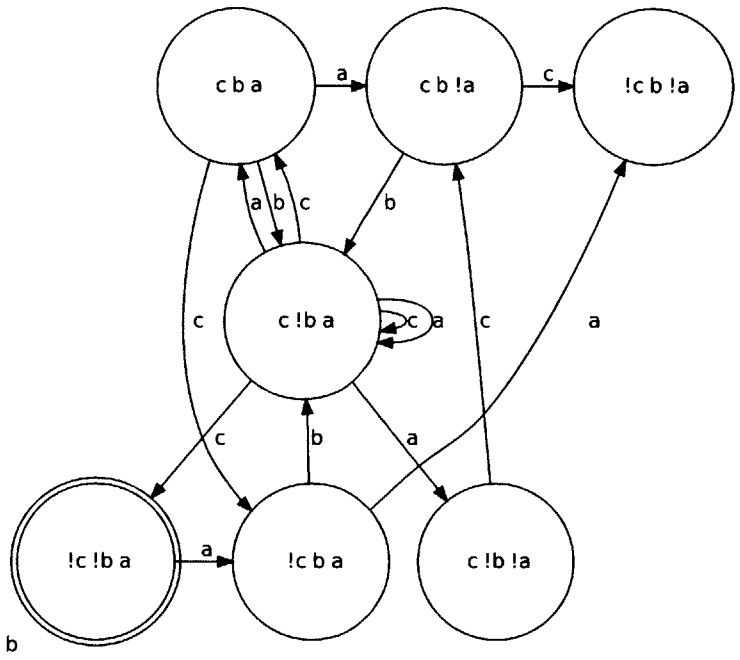


Figure 6-17: Abstract State Transition graph for BCL program in Figure 6-16.

In the first example, the application of the rule priorities let to an abstract state transition graph whose only ambiguity was a direct result of dynamic branching in the actual program. In `mkLessSimpleExample`, there is no dynamic branching, a fact which we would ideally like to be reflected the abstract state transition graph. To understand the source of the ambiguity in Figure 6-17, we need to consider the structure of rule guards. When a rule is enabled, all of its guards evaluate to true, but when a rule is disabled, only one of its guards needs to fail. In the example at hand, enabling rule `b` requires both its input FIFO (x) to be non-empty, as well as it's output FIFO (y) to not full. This means that `b` is disabled if one or both of these conditions are not met. Our choice of abstraction function does not differentiate between these cases, but the ambiguity at the node labeled "`c !b a`" is a direct result of this lack of resolution.

It is possible to increase the resolution at ambiguous nodes by further refining the abstraction function. For each ambiguous node, we examine the rules which are disabled, and for those which have a complex predicate, we extend the vocabulary of the abstraction function. Specifically, we extend the bit-vector to refer to the constituent guards of the disabled rule on which we are refining. For node "`c !b a`", this means extending

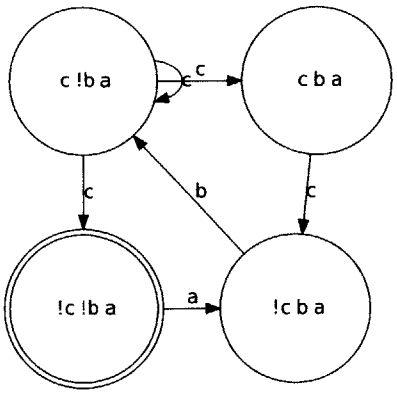


Figure 6-18: Abstract State Transition graph for BCL Program in Figure 6-16 using a priority-based scheduling strategy.

adding an entry for `x.notEmpty()` and `y.notFull()`. This is a local refinement, which proceeds exactly along the lines of graph construction. The result of this refinement, is shown in Figure 6-18. Applying a priority -based scheduling strategy results in Figure 6-19, which produces a perfect schedule. The translation from this graph to a C++ `while` loop is trivial.

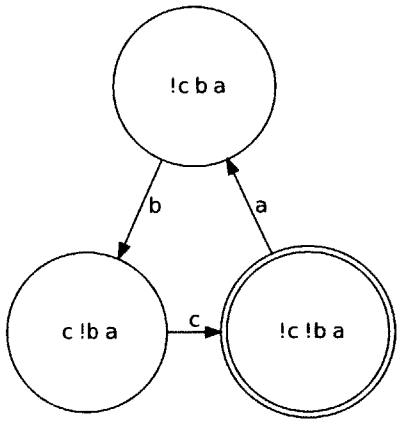


Figure 6-19: Refining the ambiguous predicates from Figure 6-18

Following this disambiguation procedure, the abstraction function may still hide some information. For example if a three-element FIFO is not-empty, it could contain one, two, or three valid tokens. In our experience, though, if this information is crucial to the enabling/disabling of a rule, it will implicitly be represented in the rule's guards.

6.3.6 Heuristics for ATS Construction

For the examples in Figure 6-2 and Figure 6-16 we constructed two graphs with varying degrees of ambiguity. In both cases, though, we were able to construct a graph which corresponded to the "best possible schedule". At any level of refinement, though, we can stop and encode the abstract state transition graph for a more efficient scheduler. This is important because the complexity of this algorithm is exponential. If we can identify heuristics which direct the graph refinement, we have a chance of producing a useful (if not perfect) result in a reasonable amount of time. The construction of the graph falls roughly into three phases: build, prune, and disambiguate. The build phase is simply the construction of the graph as described in Section 6.3.4. The pruning phase is the removal of edges by applying the priority-based scheduling strategy, and the disambiguation is the expansion of the predicate vocabulary. The worst-case size of the graph will always be exponential in the number of rules, but there are heuristics which can be applied which substantially reduce the amount of work for certain programs.

Simultaneous Build and Prune: The first is the observation that the pruning can be applied while the graph is being built. By coalescing the two phases, we avoid creating structures which will only be deleted in the next stage. Of course, the final phase will end up pruning the graph substantially too, but this requires much deeper inspection which we only want to apply locally.

Choosing the Priority: The second observation is that the priority employed by our scheduling strategy has a great impact on the graph size. The graphs in Figures 6-20 and 6-21 are produced from the same program, but with different priorities. The program is a simple four-stage pipeline separated by two-element FIFOs. The pipeline stages are each implemented by a single rule named (front-to-back) a, b, c, and d. One priority results in a graph with 8 nodes, three of which are ambiguous, while the other results in a graph with 13 nodes, four of which are ambiguous. In the absence of a user-defined scheduling priority, the compiler chooses one which reduces the graph size with the goal of reducing the amount of work. This means that pipelines are always prioritized back to front.

Topological Disambiguation: Once a graph has been constructed, the order in which the

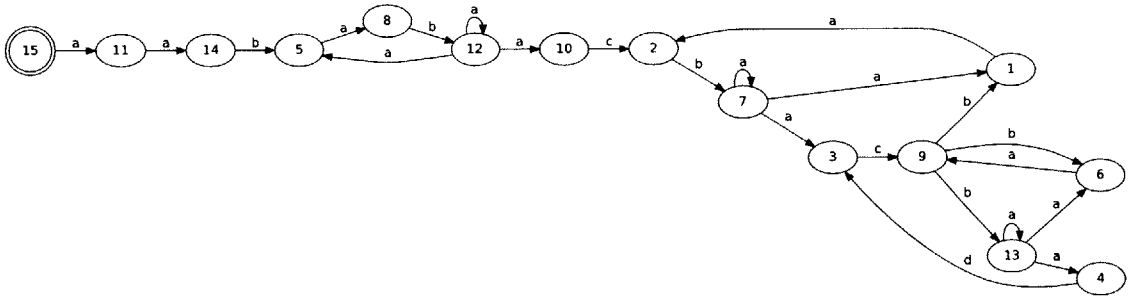


Figure 6-20: Abstract state-transition graph for a four-stage pipeline after applying the priority-based scheduling strategy [a,b,c,d].

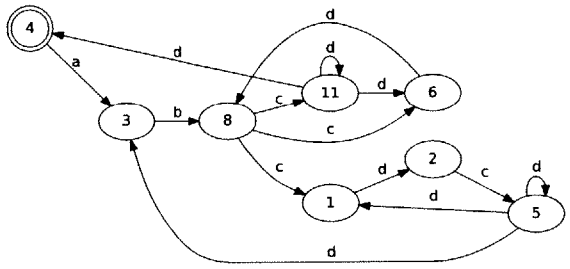


Figure 6-21: Abstract state-transition graph for a four-stage pipeline after applying the priority-based scheduling strategy [d,c,b,a].

nodes are disambiguated is important, since refinements higher in the graph topology can have the effect of removing large regions of the graph entirely. Once selected, the order in which the ambiguous predicates are enriched within a particular node appears to matter, but research into this is ongoing.

Other Strategies: Applying a scheduling strategy to reduce the graph size appears effective. We are currently looking into improvements to the current priority-based approach, placing this firmly in the "future work" category.

6.3.7 Modeling I/O

Assuming that the guards on the input and output streams are always true was a convenient way to simplify the model, and in cases where there is no relationship between the two (feedback loop), this is a safe assumption. When this is not the case, the dependency between the input and output must be modeled explicitly. This can usually be done by

inserting FIFOs to transmit control tokens between dependent I/O ports.

6.3.8 Implementing the *valid* Function using Yices

The construction of the abstract state transition graphs described previously relied on an implementation of the $valid$ function to test the legality of transitions in the graph. By definition, a labeled transition from one abstract state to another is valid iff there are members of both the source and destination sets which are connected by the rule corresponding to that edge's label. We have implemented this function using both the Z3 and Yices SMT solvers. Here we describe our use of the Yices [32] SMT solver, and how we can translate BCL to the Yices input language. To test whether the rule r1 is a valid transition between the abstract states characterized by predicate functions p0 (r2 is enabled) and p1 (r3 is enabled), we employ the Yices program shown in Figure 6-22.

```
(define as0::mkTestBench)
(define as1::mkTestBench)
(define ar2::(tuple Bool mkTestBench) (r2 as0))
(assert (select ar2 1))
(define ar3::(tuple Bool mkTestBench) (r3 as1))
(assert (select ar3 1))
(define ar1::(tuple Bool mkTestBench) (r1 as0))
(assert (= ar3 ar1))
(check)
```

Figure 6-22: The top-level Yices

The Yices code in Figure 6-22 begins by declaring the abstract states as free variables. In our translation of BCL to Yices, objects of type $Action$ are translated to functions of type $State- > (Bool, State)$. The argument has the Yices type representing program state, and the return-type of the function is a pair whose first element is a valid bit which reflects the presence of guard failures, and whose second element is the updated state. Since BCL rules are of type $Action$, they are translated to Yices functions of type $State- > (Bool, State)$. We apply the predicate functions (represented by r2 and r3) to the abstract states and bind their outputs. Asserting the truth of the valid bits on the resulting tuples guarantees that

abstract state 0 and abstract state 1 are not empty. Finally, we apply the transition rule (r1) to abstract state 0, asserting not only it's success but the equivalence of it's output to abstract state 1. If the union of these assertions are unsatisfiable, then the *valid* function returns false.

```
(define-type mkReg
 (record
   db  :: bool
   val :: bool ))

(define mkRegCtor :: (-> bool mkReg)
 (lambda (i::bool) (mk-record db::false val::i)))

(define clearDirtyBits :: (-> mkReg mkReg)
 (lambda (a::mkReg)
  (mk-record db::false val::(select a val))))

(define updateDirty :: (-> mkReg (-> mkReg mkReg))
 (lambda (a::mkReg)
  (lambda (b::mkReg)
   (if (select a db) a b))))

(define parMerge :: (-> mkReg (-> mkReg (tuple bool mkReg)))
 (lambda (a::mkReg)
  (lambda (b::mkReg)
   (mk-tuple (not (and (select a db) (select b db)))
             (if (select a db) a b)))))

(define write :: (-> bool (-> mkReg (tuple bool mkReg)))
 (lambda (x::bool)
  (lambda (st::mkReg)
   (mk-tuple true (mk-record db::true val::x)))))

(define read :: (-> mkReg bool)
 (lambda (reg::mkReg) (select reg val)))
```

Figure 6-23: Yices Functions Supporting the Definition of a Primitive Boolean Register

Because the Yices syntax is quite verbose, we give a translation of BCL Actions and Expressions (Figures 6-24 and 6-25 respectively) to Lambda expressions. With support for record types, expressions in this form can be trivially converted to Yices syntax. The

translation of BCL program state to Yices named records is direct (modulo syntax) and hence is not shown. The top-level definitions of updateDirty, clearDirtyBits, and parMerge are simple recursive traversals over the module hierarchy. We give the primitive Yices definition of a Register in Figure 6-23 to show how these three functions might be defined at the leaf nodes. All interface method names are added to the global name-space as functions whose type bears a direct relation to the method type, with the exception of an additional argument for the actual module. For example, the register write interface method, which in BCL has the type $t- > Action$ will produce the global function named "write" whose type is $(Regt)- > t- > (Bool, Regt)$. The use of yices has been fully automated and integrated into the BCL compiler in the rule-scheduling phase.

TA :: BCL-Action → (State → (Bool, State))

TA $[\![r := e_0]\!] = \lambda$ s. $(g,$ s$\{r =$ write $(r$ s$)\ e_1\})$
where $(g, e_1) =$ (**TE** $[\![e_0]\!]$) s

TA $[\![$ **if** $p_0\ a_0]\!] = \lambda$s. $(g_1 \wedge (\neg p_1 \vee g_2),$ if p_1 then s_1 else s)
where $(g_1, p_1) =$ (**TE** $[\![p_0]\!]$) s
$(g_2, s_1) =$ (**TA** $[\![a_0]\!]$) s

TA $[\![a_0$ **when** $e_0]\!] = \lambda$s. $(g_1 \wedge g_2 \wedge e_1, s_1)$
where $(g_1, s_1) =$ (**TA** $[\![a_0]\!]$) s
$(g_2, e_1) =$ (**TE** $[\![e_0]\!]$) s

TA $[\![a_1 \mid a_2]\!] = \lambda$s. let s_1 = clearDirtyBits s in
let (g, s_4) = parMerge $s_2\ s_3$ in
$(g_1 \wedge g_2 \wedge g,$ updateDirty s $s_4)$
where $(g_1, s_2) =$ (**TA** $[\![a_1]\!]$) s_1
$(g_2, s_3) =$ (**TA** $[\![a_2]\!]$) s_1

TA $[\![a_1$; $a_2]\!] = \lambda$s.$(g_1 \wedge g_2, s_2)$
where $(g_1, s_1) =$ (**TA** $[\![a_1]\!]$) s
$(g_2, s_2) =$ (**TA** $[\![a_2]\!]$) s_1

TA $[\![$ t = e_0 **in** $a_0]\!] = \lambda$s.let $tg = g_1$ in
let $tb = e_2$ in (g_2, s')
where $(g_1, e_1) =$ (**TE** $[\![e_0]\!]$) s
$(g_2, s') =$ (**TA** $[\![a_0[(tb$ **when** $tg)/t]]\!]$) s
tg, tb are fresh names

TA $[\![$ **loop** $e_0\ a_0]\!] =$ loopUntil (**TE** $[\![e_0]\!]$) (**TA** $[\![a_0]\!]$)
where loopUntil = λfe.λfa.λ s.
let (g1, e1) = fe s in
let (g2, s_1) = fa s in
let (g3, s_2) = loopUntil fe fa s1 in
(g1 $\wedge$ ($\neg$ e1 $\vee$ (g2 $\wedge$ g3)), if e1 then s_2 else s)

TA $[\![$ **localGuard** $a_0]\!] = \lambda$s.(True, if g then s_1 else s)
where $(g, s_1) =$ (**TA** $[\![a_0]\!]$) s

TA $[\![$ m.g(e_0) $]\!] = \lambda$s. let $(g_1, m_1) =$ g m e_1 in $(g_0 \wedge g_1,$ s$\{$m = $m_1)\}$
where $(g_0, e_1) =$ (**TE** $[\![e_0]\!]$) s

Figure 6-24: Translating BCL Actions to Yices-compatible Lambda Expressions

TE :: BCL-Expr $\rightarrow$ (State $\rightarrow$ (Bool, t))

TE $[\![$ r $]\!]$ =λs. (True, read (r s))
TE $[\![$ c $]\!]$ =λs. (True, c)
TE $[\![$ t $]\!]$ =λs. (True, t)

TE $[\![e_1$ **op** $e_2]\!] = \lambda$s.$(g_1 \wedge g_2, e'_1$ **op** $e'_2)$
where (g_1, e'_1) = (**TE** $[\![e_1]\!]$) s
(g_2, e'_2) = (**TE** $[\![e_2]\!]$) s

TE $[\![e_b$ **?** e_t **:** $e_f]\!] = \lambda$s. $(g_b \wedge$ (if e'_b then g_t else $g_f)$, if e'_b then e'_t else $e'_f)$
where (g_b, e'_b) = (**TE** $[\![e_b]\!]$) s
(g_t, e'_t) = (**TE** $[\![e_t]\!]$) s
(g_f, e'_f) = (**TE** $[\![e_f]\!]$) s

TE $[\![e$ **when** $e_g]\!] = \lambda$s. $(g_g \wedge e'_g \wedge g_e, e')$
where (g_e, e') = (**TE** $[\![e]\!]$) s
(g_g, e'_g) = (**TE** $[\![e_g]\!]$) s

TE $[\![t = e_1$ **in** $e_2]\!] = \lambda$s.(let $tg = g_1$ in (let $tb = e'_1$ in $(g_2, e'_2)))$
where (g_1, e'_1) = (**TE** $[\![e_1]\!]$) s
(g_2, e'_2) = (**TE** $[\![e_2[(tb$ **when** $tg)/t]]\!]$) s
tg, tb are fresh names

TE $[\![$ m.f(e_0) $]\!] = \lambda$s.let (g_1, e_2) = f (m s) e_1 in $(g_0 \wedge g_1$, e)
where (g_0, e_1) = (**TE** $[\![e_0]\!]$) s

Figure 6-25: Translating BCL Expressions to Yices-compatible Lambda Expressions

Chapter 7

Related Work

There is a substantial body of work, both academic and commercial, relevant to various aspects of hardware-software codesign. We discuss them in this chapter.

Implementation-agnostic parallel models: There are several parallel computation models whose semantics are agnostic to implementation in hardware or software. In principle, any of these can provide a basis for hardware-software codesign.

Threads and locks are used extensively in parallel programming and also form the basis of SystemC [47] – a popular C++ class library for modeling embedded systems. While these libraries provide great flexibility in specifying modules, the language itself lacks proper compositional semantics, producing unpredictable behaviors when connecting modules. Synthesis of high-quality hardware from SystemC remains a challenge.

Dataflow models, both at macro-levels (Kahn [42]) and fine-grained levels (Dennis [30], Arvind [9]), provide many attractive properties but abstract away important resource-level issues that are necessary for expressing efficient hardware or software. Nevertheless dataflow models where the rates at which each node works are specified statically have been used successfully in signal processing applications [46, 60].

Synchronous dataflow is a clean model of concurrency based on synchronous clocks, and forms the basis of many programming languages (*e.g.,* Esterel [14], Lustre [37], Signal [23], Rapide [49], Shim [33], Polysynchrony [59]). Scade [5], a commercial tool for designing safety-critical applications, is also based on this model.

We have chosen *guarded atomic actions* (or rules) as the basis for *BCL*. All legal behaviors in this model can be understood as a series of atomic actions on a state. This model was used by Chandy and Misra in Unity [19] to describe software, and then by Hoe and Arvind to generate hardware [40]. Dijkstra's Guarded Commands [31] and Lynch's IO Automata [50] are also closely related. BCL extends the idea of multiple clock domains [24] from Bluespec SystemVerilog (BSV) [17] to specify how a design should be split between hardware and software. Hardware compilation from BCL is a straightforward translation into BSV, whose subsequent compilation to Verilog is a mature and proven technology [17, 40]. Guarded atomic actions also provide a good foundation to build analysis and verification tools [26].

Generation of software from hardware descriptions: Hardware description languages (HDLs) like Verilog and VHDL provide extremely fine-grain parallelism but lack an understandable semantic model [52] and are also impractical for writing software. Nevertheless, these languages can compile to a software simulator, which can be viewed as a software implementation. Popular commercial products like Verilator [65] and Carbon [1] show significant speedup in the performance of these simulators, though the requirement to maintain cycle-level accuracy (at a gate-level) is a fundamental barrier; the performance is often several factors slower than natural software implementations of the same algorithm.

The Chinook [21, 22] compiler addressed an important aspect of the hardware/software interface problem, by automating the task of generating a software interface from the RTL description and the timing information of the hardware blocks.

Lastly, Bluespec's Bluesim [17] can exploit the fact that the cycle-level computation can be represented as a sequence of atomic actions. This permits dramatic improvement in performance but the underlying cost of cycle-accuracy remains.

Generation of hardware from sequential software specifications: To avoid the burden of using low-level HDLs, the idea of extracting a hardware design from a familiar software language, *e.g.*, C, Java, or Haskell, has great appeal. Liao et al. [48] present one of the earliest solutions to mitigate the difficulty of the hardware-software interface problem by describing the interface in C++ and generating an implementation using hand coded

libraries that interact with automatically synthesized hardware. Many systems like CatapultC [51], Pico Platform [58], or AutoPilot [11] have been effective at generating some forms of hardware from C code. However, constructing efficient designs with dynamic control can be very hard, if not impossible, using such tools [7].

A related effort is the Liquid Metal project [41] which compiles an extension of Java into hardware. It lets the programmer specify parts of the program in a manner which eases the analysis required for efficient hardware generation. In contrast to BCL which relies on explicit state and guarded atomic actions, Liquid Metal exploits particular extensions to the Java type system.

Frameworks for simulating heterogeneous systems: There are numerous systems that allow co-simulation of hardware and software modules. Such systems, which often suffer from both low simulation speeds and improperly specified semantics, are typically not used for direct hardware or software synthesis.

Ptolemy [18] is a prime example of a heterogeneous modeling framework, which concentrates more on providing an infrastructure for modeling and verification, and less on the generation of efficient software; it does not address the synthesis of hardware at all. Metropolis [12], while related, has a radically different computational model and has been used quite effectively for hardware/software codesign, though primarily for validation and verification rather than the synthesis of efficient hardware.

Matlab and Simulink generate production code for embedded processors as well as VHDL from a single algorithmic description. Simulink employs a customizable set of block libraries which allow the user to describe an algorithm by specifying the component interactions. Simulink does allow the user to specify modules, though the nature of the Matlab language is such that efficient synthesis of hardware would be susceptible to the same pitfalls as C-based tools. A weakness of any library-based approach is the difficulty for users to specify new library modules.

In summary, while all these frameworks may be effective for modeling systems, we do not believe they solve the general problem of generating efficient implementations.

Algorithmic approaches to hardware/software partitioning: There is an extensive body

of work which views hardware-software partitioning as an optimization problem, similar to the way one might look at a graph partitioning problem to minimize communication [8, 20]. The success of such approaches depends upon the quality of estimates for various cost functions as well as the practical relevance of the optimization function. Since these approaches do not generate a working hardware/software design, they need to make high-level approximations, often making use of domain-specific knowledge to improve accuracy. Such analysis is complementary to real hardware-software codesign approaches.

Infrastructures for FPGA Portability: There is a lot of work in this area. To begin with, very FPGA vendor distributes a suite of tools which claims to support common abstractions over a range of different products. Vivado and Platform Studio from Xilinx are good examples of this. Other more academic approaches to this are LEAP [] from Emer et. al., and BORPH, from Kwok-Hay So et.al. Each solution distinguishes itself primarily by the choice of abstractions, which makes comparing them a matter of taste as much as anything else. In this sense, the Platform Support we have built for our own purposes is very similar to LEAP and BORPH. Our solution distinguishes itself from all others we are aware of by its tight integration with the BCL language.

Chapter 8

Conclusion and Future Work

In this thesis, we proposed a structured methodology for the integration of specialized hardware into an application running in software. The incremental approach begins simply by reorganizing the application into a structured dataflow graph. Restructuring an application in this manner is not trivial, but once this form is achieved, our interface compiler can automatically generate the hardware-software interface (as well as the HW-HW and SW-SW dataflow interfaces). Besides enabling the automatic synthesis of interfaces, the encapsulation inherent to this structure greatly eases the task of moving functionality (at the granularity of nodes in the graph) from hardware to software or vice versa.

Within the dataflow framework, the challenge of re-implementing the functionality being moved between hardware and software persists. This leads to the second contribution of the thesis, where instead of requiring precise interface specification a priori, the user specifies all or parts of the design, *including* the HW/SW decomposition, in a single language. The resulting specification can be used to generate efficient implementations of computational nodes within the structured dataflow graph as well as the edges which encapsulate the communication. Using BCL greatly reduces the partitioning and reimplementation effort, allowing the programmer to concentrate exclusively on the performance implications of the partitioning choices. BCL concedes nothing in the quality of hardware, and little in the quality of software that is generated.

The final contribution of this thesis is an SMT-based approach to inferring the high-level structure of HW-centric BCL Rule specifications. When applied successfully, this

technique allows us to generate *truly* "no-compromise" software and hardware from the same code, thereby fulfilling the ultimate promise of single language HW-SW codesign.

A complete system has implemented to realize this goal, including an interface compiler, a BCL compiler, both software and hardware run-times, and support on both the Zynq and ML507 platforms from Xilinx. We have illustrated our approach using examples written both in C++/BSV and BCL.

Future Work: The work presented in this thesis is already useful, but there are many ways it could be extended or enriched. The following list enumerates a few of the author's favorites:

1. **Standard Benchmark Suite:** In Chapter 2, we employed a somewhat ad hoc methodology for evaluating the impact of specialized hardware in the throughput of various applications. Some of the reasons for this are unavoidable (as cited, so much as platform dependent), but compiling a standard benchmark suite would be extremely helpful in identifying which benchmarks are important. Where applicable, benchmark implementation effort can also be avoided.

2. **A More Efficient procedure for inferring structure:** Chapter 6 proposed an SMT-based procedure for inferring high-level dataflow in rule-based systems. Improving on the efficiency of this is necessary for it's widespread use. It is also possible that alternate approach not considered in this thesis might accomplish the same goal much more efficiently.

3. **DSLs with more structure as opposed to rules:** BCL is higher-level and easier to use than the majority of HDLs, but when compared to almost every software language, it is quite cumbersome. The argument made in favor of rules it that they permit the programmer to express the fine-grained parallelism *required* for efficient hardware. Precisely because of this flexibility (lack of structure) inferring efficient software from a collection of rules is difficult. In most of the designs, we could have used a more restrictive language without a loss in hardware quality. Future work into less general (more targeted) rule based languages might not only make the programming task easier, but the compiler's burden as well.

4. **Multi-Threaded SW:** Though we mentioned it briefly in Chapter 6, the BCL implementation to date does not make use of multi-threading. Exploring when this is useful and how best to implement the SW-SW dataflow edges would be interesting.

5. **SW-Friendly Interfaces:** The abstraction of guarded FIFOs is ideal for compositional hardware implementations since the introduction of registers is often required to avoid long combinational paths between blocks and adding guard logic to them is relatively low cost. Due to the way in which rules are schedules, the FIFO guards enforce dataflow dependency without adding to the cost of execution. This style of buffering between modules is not as good for software, since it breaks up the long chains of method calls which can be more efficiently compiled by a C++ Compiler. The author is interested to explore different styles of guarded interfaces, perhaps using explicit reservation-style functionality, which permits a more software friendly way of composing modules without sacrificing safety.

Bibliography

[1] Carbon Design Systems Inc. `http://carbondesignsystems.com`.

[2] Ogg Vorbis. `http://www.vorbis.com`.

[3] Standard Performance Evaluation Corporation. `http://www.spec.com`.

[4] The Embedded Microprocessor Benchmark Consortium. `http://www.eembc.com`.

[5] Parosh Aziz Abdulla and Johann Deneux. Designing safe, reliable systems using scade. In *In Proc. ISoLA 2004*, 2004.

[6] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 25–28, New York, NY, USA, 2011. ACM.

[7] A. Agarwal, Man Cheuk Ng, and Arvind. A comparative evaluation of high-level hardware synthesis using reed-solomon decoder. *Embedded Systems Letters, IEEE*, 2(3):72 –76, sept. 2010.

[8] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Trans. Des. Autom. Electron. Syst.*, 10:136–156, January 2005.

[9] Arvind and R.S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture . *Computers, IEEE Transactions on*, 39(3):300 –318, March 1990.

[10] Lennart Augustsson, Jacob Schwarz, and Rishiyur S. Nikhil. Bluespec Language definition, 2001. Sandburst Corp.

[11] AutoESL Desisgn Technologies, Inc. `http://www.autoesl.com`.

[12] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36:45–52, 2003.

[13] J.S. Beeckler and W.J. Gross. Fpga particle graphics hardware. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 85–94, 2005.

[14] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency*, pages 389–448, 1984.

[15] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[16] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[17] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[18] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[19] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[20] Karam S. Chatha and Ranga Vemuri. Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *CODES*, pages 42–47, 2001.

[21] Pai Chou, Ross Ortega, and Gaetano Borriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *In Proceedings of the International Conference on Computer Aided Design*, pages 488–495, 1992.

[22] Pai H. Chou, Ross B. Ortega, and Gaetano Borriello. The chinook hardware/software co-synthesis system. In *In International Symposium on System Synthesis*, pages 22–27, 1995.

[23] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, pages 73–82, New York, NY, USA, 2006. ACM.

[24] Ed Czeck, Ravi Nanavati, and Joe Stoy. Reliable Design with Multiple Clock Domains. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[25] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[26] Nirav Dave, Michael Katelman, Myron King, Jose Meseguer, and Arvind. Verification of Microarchitectural Refinements in Rule-based systems. In *MEMOCODE*, Cambridge, UK, 2011.

[27] Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. A design flow based on modular refinement. In *Formal Methods and Models for Codesign (MEMOCODE 2010)*.

[28] Nirav Dave, Michael Pellauer, Steve Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Napa, CA, 2006.

[29] Nirav H. Dave. *A Unified Model for Hardware/Software Codesign*. PhD thesis, MIT, Cambridge, MA, 2011.

[30] Jack B. Dennis, John B. Fosseen, and John P. Linderman. Data flow schemas. In *International Sympoisum on Theoretical Programming*, 1972.

[31] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8), 1975.

[32] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006.

[33] Stephen A. Edwards and Olivier Tardieu. Shim: a deterministic model for heterogeneous embedded systems. *IEEE Trans. VLSI Syst.*, 14(8):854–867, 2006.

[34] Thomas Esposito, Mieszko Lis, Ravi Nanavati, Joseph Stoy, and Jacob Schwartz. System and method for scheduling TRS rules. United States Patent US 133051-0001, February 2005.

[35] K. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J. Emer. Leveraging latency-insensitivity to ease multiple fpga design. In *FPGA*, February 2011.

[36] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind, and Joel S. Emer. Leveraging latency-insensitivity to ease multiple fpga design. In *FPGA*, pages 175–184, 2012.

[37] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305 –1320, sep 1991.

[38] Eric Hamilton. JPEG File Interchange Format v1.02. `http://www.w3.org/`, 1992.

[39] James C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, MIT, Cambridge, MA, 2000.

[40] James C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 23(9), September 2004.

[41] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, Berlin, Heidelberg, 2008.

[42] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[43] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *ICCAD*, 2008.

[44] Myron King, Nirav Dave, and Arvind. Automatic generation of hardware/software interfaces. ASPLOS '12, New York, NY, USA, 2012. ACM.

[45] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[46] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[47] S. Y. Liao. Towards a new standard for system level design. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 2–7, San Diego, CA, May 2000.

[48] Stan Y. Liao, Steven W. K. Tjiang, and Rajesh K. Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *DAC*, pages 70–75, 1997.

[49] David C. Luckham. Rapide: A language and toolset for causal event modeling of distributed system architectures. In *WWCA*, 1998.

[50] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[51] Mentor Graphics. *Catapult-C Manual and C/C++ style guide*, 2004.

[52] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010.

[53] Daniel L. Rosenband. *A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions*. PhD thesis, MIT, Cambridge, MA, 2005.

[54] Alessandro Forin Scott Sirowy. Where's the beef? why fpgas are so fast. MSR-TR-2008-130, September 2008.

[55] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003.

[56] Armando Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009.

[57] Martin Sulzmann, Jeremy Wazny, and Peter J. Stuckey. A framework for extended algebraic data types. In *FLOPS*, pages 47–64, 2006.

[58] Synfora. PICO Platform. `http://www.synfora.com/`.

[59] Jean-Pierre Talpin, Christian Brunette, Thierry Gautier, and Abdoulaye Gamatié. Polychronous mode automata. In *EMSOFT*, pages 83–92, 2006.

[60] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*. Springer, 2002.

[61] K.H. Tsoi, K.H. Lee, and P. H W Leong. A massively parallel rc4 key search engine. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 13–21, 2002.

[62] International Telecoms Union. JPEG Specification. `http://www.w3.org/Graph- ics/JPEG/itu-t81.pdf`, 1992.

[63] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. Global critical path: A tool for system-level timing analysis. In *DAC*, pages 783–786, 2007.

[64] Muralidaran Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *MEMOCODE*, pages 171–180, 2009.

[65] W. Snyder and P. Wasson, and D. Galbi. Verilator. `http://www.veripool.com/verilator.html`, 2007.

[66] K. Whitton, X.S. Hu, C.X. Yu, and D.Z. Chen. An fpga solution for radiation dose calculation. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 227–236, 2006.

[67] Stephen B. Wicker and Vijay Bhargava. *Reed-Solomon Codes and Their Applications*. IEEE Press, New York, 1994.