An Efficient Sequential BTRS Implementation

by

Myron Decker King

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author .... [illegible]
Department of Electrical Engineering and Computer Science
February 1, 2009

Certified by .... [illegible]
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible]
Terry P. Orlando
Chairman, Department Committee on Graduate Students

An Efficient Sequential BTRS Implementation

by

Myron Decker King

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis describes the implementation of BTRS, a language based on guarded atomic actions (GAA). The input language to the compiler which forms the basis of this work is a hierarchical tree of modules containing state, interface methods, and rules which fire atomically to cause state transitions. Since a schedule need not be specified, the program description is inherently nondeterministic, though the BTRS language does allow the programmer to remove nondeterminism by specifying varying degrees of scheduling constraints. The compiler outputs a (sequential) single-threaded C implementation of the input description, choosing a static schedule which adheres to the input constraints. The resulting work is intended to be used as the starting point for research into efficient software synthesis from guarded atomic actions, and ultimately a hardware inspired programming methodology for writing parallel software. This compiler is currently being used to generate software for a heterogeneous system in which the software and hardware components are both specified in BTRS.

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

Acknowledgments

I would like to acknowledge the contributions of Nirav Dave with whom I collaborated on this work. Without his valuable insight, and technical excellence, this project might never have gotten off the ground. Equally important was the input and guidance of my advisor Arvind. This research was funded in part by Nokia Inc.

Contents

List of Figures

Chapter 1

Introduction

Concurrent programming is challenging even for experienced programmers. This is due in part to the fact that limited progress has been made in finding better ways of expressing parallelism in software. New programming models are required which allow programmers to exploit safely and effectively the ever increasing parallelism made available by new generations of processors and other programmable substrates. Throughout its history, hardware design has focused on parallel systems and could therefore provide a good model for the development of more generic concurrent software programming techniques. The problem with most hardware languages is that they are too low level (close to gate representations) for a natural translation in to efficient software.

In order to illustrate the importance of efficient software AND hardware generation, consider the process of writing device drivers or embedded software. Suppose that a system composed of both software components (running on a CPU) and hardware components (connected by a Bus) must be built. The software interacts with the hardware through some predefined interface. Often the initial choice of design partition will later prove to be less than ideal, requiring substantial change to both the hardware and software. If the hardware and software were written in the same language, repartitioning the design would require only a change in the designation of compilation targets for the modules crossing the boundary.

Solutions to the two aforementioned challenges are related insofar as both would both make use of the generation of efficient software from a language which employs successful techniques borrowed from hardware design. The work in this thesis is based on the notion that a hardware-design-inspired methodology will make it easier to write parallel software that is less susceptible to pitfalls such as deadlock, and inconsistency, which plague programs written in sequential languages (C, C++, Java, etc.) using synchronization primitives such as semaphores and locks. The proposed methodology is based on the use of Guarded Atomic Actions, which combined with distributed control and guarded interfaces, will serve as a starting point for the generation of efficient parallel software.

The solution described in this thesis is a single-threaded, sequential implementation of the formal semantics of BTRS[3], a language closely related to the commercially available Bluespec System Verilog (BSV). It is believed that this could become an effective tool both in stand-alone software environments and in integrated HW/SW codesign scenarios.

1.1 Hardware-Inspired Methodology

The following principles have been found to be very effective in hardware design: [7] [5]

1. Resource Awareness: In order to share resources effectively, they cannot be virtualized. The explicit multiplexing of resources requires domain specific knowledge which is extremely difficult, often impossible, to derive automatically. One fundamental difference between a hardware and software implementation of an algorithm is whether the data is moved over the algorithm (HW) or the algorithm is moved over the data (SW). When implementing an algorithm in software, it is often most efficient to place the data in system memory and iterate over it, mutating it until the memory contains the final result. This is referred to as "moving the algorithm over the data". A hardware design, on the other hand, might consist of one or more FSMs for each stage of the algorithm, and some memory where the data is stored. Each stage of computation modifies the data and passes it to the next stage of algorithm, with the

first stage fetching from and the final stage storing to memory. This approach is clearly "moving the data through the algorithm".

2. Distributed Control and Guarded Interfaces: It is important that control decisions are not left up to the compiler, but are instead encoded explicitly into the system description. Intertwining control and data-path has proven advantageous. This approach is different from most parallel languages which describe only local interactions. We use guarded interfaces to describe these aspects of a program, making it easier to avoid intractable concurrency errors.

3. User Control of Parallel exploitation: Many sequential algorithmic control structures are too hard for compilers to parallelize. A language which requires explicit description of parallel control will allow the programmer to easily expose parallelism, and provide clear compositional semantics.

1.1.1 Guarded Atomic Actions

One of the biggest difficulties in writing parallel code is the lack of composability. Generally, through painstaking manual effort, parallel libraries can be written, but when more than one parallel libraries are composed, the semantics are generally unclear and the results are often less than satisfactory. The reason for this is that the composable abstractions used in software, such as objects and methods, lend themselves well to sequential composition, but have no clear parallel semantics. Ideally a language should define clear compositional semantics in both parallel and sequential cases.

Problems in current approaches to parallelism also arise because SW abstractions tend to hide resources (cpu, memory, etc.), and rely on the compiler and runtime to multiplex their use. Modern compiler techniques fail to do this efficiently on all but the simplest (embarrassingly parallel) of algorithms. A possible solution to this problem is to manually implement space multiplexing, which is the practice of assigning different parts of a parallel computation to different physical resources (computation nodes). This breaks down when the number of resources (cpu cores) do not equal the number of threads. While improving resource usage, this approach does nothing to alleviate the previously mentioned problems of using an inherently sequential language to describe a parallel process. Space multiplexing also raises the specter of interprocess communication, which can be very expensive and is difficult for compiler tools to analyze without a lot of auxiliary information (production and and consumption rates, etc.)

Guarded Atomic Actions are a substantially higher level of abstraction than standard RTL and have been shown to be an effective tool in designing hardware, a task which requires both clear parallel semantics as well as resource awareness. It is possible that they can prove just as effective in the generation of parallel software.

1.2 A New Way of Thinking

Thinking about parallel programming as a synthesis process from a set of modules with proper (guarded) interfaces and clear compositional semantics may point to a new paradigm in which the programmer can easily express an algorithm that avoids obscuring parallelism. The need for a software language encapsulating our hardware-inspired methodology gave rise to the definition of the BTRS language. The semantics of BTRS are an extension of BSV.

1.3 Intended Application

An immediate use for this technology can be found in the automatic generation of device drivers in HW/SW codesign environments. The eventual goal is to provide an unchanging hardware abstraction layer to software developers and allow hardware developers to freely change their implementation without requiring constant manual updates to the low-level driver software. The gap between what the user program "sees" and what the hardware implements would be automatically generated by this compiler.

This compiler could also be used for the generation of software designed to run in multi-core environments. This is the subject of future research, with many exciting possibilities in the area of rule scheduling.

1.4 Organization

This thesis begins with a brief sketch of the compilation challenges, and is followed by a formal description of the semantics of the BTRS language. A detailed discussion of the compiler implementation along with a specification of the syntax directed compilation scheme is given. The paper ends with a description of the application used to evaluate the compiler, an evaluation section, and a brief discussion of related work. The compiler is implemented in Haskell and generates C/C++. A working knowledge of Haskell, C/C++ and Bluespec System Verilog is assumed.

Chapter 2

Compiling A System of Rules

In our methodology, programs consist of state (of type S) and rules to modify the state (pure functions of type $S \rightarrow S$). At the highest level, compiling a piece of software specified as a collection of rules and modules (state) is quite simple: assemble all the rules into a list and create a main function which iterates over the list, executing the rules on the state sequentially in their listed order. In this fashion, it should be apparent that the quality of the generated software is directly related to the efficiency of the rule specifications.

2.1 Single Rule Compilation

Since rules are pure functions which mutate the program state, one of the challenges of compiling them efficiently (in isolation) is to minimize the amount of temporary state required to implement the correct semantics. There is an additional complication in that the rules being considered here have two significant features not encountered in standard sequential programming languages. The rules are described in detail in the following sections.

2.1.1 Actions Composed in Parallel

All state updates (actions) within a rule are composed in parallel. Furthermore, the language semantics imply that rules are executed in complete isolation producing effects that are invisible to other rules in execution. Lastly, all state is read in parallel at the beginning of the rule execution and all updates are committed in parallel at the end of execution. Consider the following rule containing two register updates:

```
rule A;
  r1 <= r2+1;
  r2 <= r1+1;
endrule
```

Translating this rule into sequential C requires the introduction of shadow state in order to achieve the parallel semantics:

```
void A(){
  t1 = r1.read();
  t2 = r2.read();
  r1 = t2+1;
  r2 = t1+1;
}
```

In addition to simple registers, the notion of state can be augmented with hierarchically structured modules with interface methods that mutate the state within the module or return a value. Since modules can be flattened, they do not add or or subtract from the expresivity of the previous picture composed exclusively of rules and simple registers. This will become an important part of the language since it allows the programmer to partition state and functionality and allocate resources accordingly. Consider the following rule consisting of two user-module action-method invocations, and the corresponding definitions of the actions:

```
rule B;
  sub_mod.action1();
  sub_mod.action2();
endrule
method SubMod::action1();
  r1 <= r2+1;
  r2 <= r1+2;
endmethod
method SubMod::action2();
  r3 <= r2+1;
  r4 <= r3+1;
endmethod
```

A straightforward translation of rule B requires that sub_mod be shadowed since the actions must appear to be executed in parallel. In addition, the definition of sub_mod must be augmented with a copy constructor and some notion of parallel merging, so that the results of executing action1 and action2 in parallel can be correctly exposed. The action methods are translated to C in a similar fashion as rule A in the previous example. Below is a translation of rule B, with the assumption that sub_mod is of type SubMod:

```
void B(){
  shadow1 = SubMod(sub_mod);
  shadow2 = SubMod(sub_mod);
  shadow1.action1();
  shadow2.action2();
  sub_mod.parMerge(shadow1);
  sub_mod.parMerge(shadow2);
}
```

This example shows that implementing parallel semantics may increase the amount of required shadow state dramatically, since state must be shadowed at each level of the module hierarchy. Performing the analysis to reduce the amount of shadow state is a major focus of the compilation effort.

2.1.2 Guarded Actions

Each rule and method is guarded by a boolean condition and can only "fire" when that condition is true. There is no partial firing; if any guard within a rule fails, the entire rule will fail. Action guards are not to be confused with conditional execution. A guard failure on any constituent action of a composite action will cause the entire composite action itself to fail. If an action fails, it does not update any state. Consider rule B defined in the previous example. The given translation is not quite correct, since it does not take into account the possibility that action1 or action2 could fail. The translation below takes this possibility into account and augments the actions and rule with guards. The keyword **when** is used to denote a guard condition.

```
rule B when cond1;
  sub_mod.action1();
  sub_mod.action2();
endrule
method SubMod::action1() when cond2;
  r1 <= r2+1;
  r2 <= r1+2;
endmethod
method SubMod::action2() when cond3;
  r3 <= r2+1;
  if (a) then r4 <= r3+1; endif
endmethod
```

The updatede C translation is given below. The implementation of action2 illustrates the difference between guard failure and normal conditional execution:

```
void B(){
  if (cond1){
    try{
      shadow1 = SubMod(sub_mod);
      shadow2 = SubMod(sub_mod);
      shadow1.action1();
      shadow2.action2();
      t1 = r1.read();
      t2 = r2.read();
    }catch (error e){throw e;}
    sub_mod.parMerge(shadow1);
    sub_mod.parMerge(shadow2);
  }else{throw (guard_failure);}
}
void SubMod::action1(){
  ...
}
void SubMod::action2(){
  if (cond2){
    t1 = r2.read();
    t2 = r3.read();
    r3 = t2+1;
    if (a) r4 = t1+1;
  }else{throw (guard_failure);}
}
```

This translation makes use of try/catch blocks to avoid partial state updates due to guard failures. There is a large overhead assciated with these constructs in C++, making this quite an inefficient translation. Optimizing these cases is an important part of the compilation effort.

2.2 Multiple Rule Compilation

Finding an efficient schedule within the constraints specified by the programer is an equally important aspect of the software generation process. Derived rules create new rules by composing other rules either in parallel or as a sequence. Such constructs present unique challenges as well as optimization oportunities, as inter-rule optimizations present themselves. As of the writing of this thesis, little effort has been spent on this aspect, though future work will likely focus on it more.

Chapter 3

BTRS: A Language of Guarded Atomic Actions

BTRS a language of guarded atomic actions based on the hardware description language Bluespec System Verilog (BSV). It has a lot in common with transactional memory systems, although 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. Usng 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. Optmistic 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 BTRS programs as well. BTRS guarded atomic actions differ from transactional memories in several important ways:

1. BTRS 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

subactions. 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 BTRS.

2. BTRS 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 BTRS, which roughly corresponds to BSV after "static elaboration," *i.e.,*after typechecking, constant propagation and module instantiations. To make the language more suitable for software specification, BTRS adds a sequential connective in addition to the parallel connective. The semantics have been described previously [4] in the context of hardware description. For the sake of completeness, the complete language description has been included. The rest of this chapter is taken directly from the original paper [4] with a few modifications. The figures (3-1, 3-2, 3-3, 3-4, 3-5, 3-2, 3-6, and 3-7) are included without modifications.

3.1 BTRS Syntax

A grammar for BTRS is given in Figure 3-1. A BTRS program consists of 2 parts: a set of state elements and a set of guarded atomic actions or *rules* which represent the state changes. State consists of modules, which are accessed through *interface methods*. Modules can contain their own internal state, as well as rules, which modify state internal to that module. The language provides some primitive modules, such as the Register, which form the basis for all program state.

It should be apparent to the reader that all communication between processes in a language describe in this manner is explicit by definition. This turns out to have important ramifications in the compiler implementation.

```
m ::=  Module name
       [Register r v]       // Regs w/ initial values
       [Rule R a]           // Rules
       [ActMeth g λx.a]     //Action method
       [ValMeth f λx.e]     //Value method
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
   ||  m.g(e)               // Action Methcall 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 Methcall f of m
op ::= && | || | ...        // Primitive operations
```

Figure 3-1: BTRS Grammar for a Module

3.2 Semantics of Rule Execution in BTRS

Every action or rule in BTRS modifies the state deterministically. The nondeterminism in a description is introduced by the choice in the order of execution of these rules. The range of behaviors that a collection of modules and rules can produce is succinctly described in

Repeatedly:

1. Choose a rule in some module to execute
2. Compute U, the set of register updates, by evaluating the rule's action according to the rules given in Figure 3-3.
3. Update all the registers according to U.

Figure 3-2: BTRS Execution Procedure

the Figure 3-2. Notice that this procedure uses nondeterministic choice which may affect the final behavior, making a BTRS program more like a specification. How we resolve this nondeterminism in the design is very important for the quality of the implementation. In cases where performance is important, the designer may wish to define the scheduler. In less important cases, designers can leave the decision to the compiler or merely provide hints.

We present the operational semantics of a rule execution in BTRS using SOS-style evaluation rules (Figures 3-3, 3-4, and 3-5), where $\twoheadrightarrow$ means either expression evaluation or the effect of an action. The meaning of each composite atomic action will be explained in terms of its constituent actions.

Let S represent the values of all the registers before the rule executes. The effect of executing an atomic action will be represented by U, the set of register updates implied by the execution. Conflicting updates to the same register produce a *dynamic error*. Our system can easily handle dynamic errors, but doing so would clutter the presentation. For the purposes of this thesis, we will assume that the sufficient static analysis has been applied to all system to prevent dynamic errors from occurring.

The semantic machine is incomplete in the sense that there are cases where the execution gets *stuck* because none of the rules in Figure 3-3 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.

So far, there doesn't seem to be anything particularly novel about BTRS. The less

Action Rules:

reg-update
$$\frac{\langle S,U,B\rangle \vdash e \twoheadrightarrow v,\ v \neq \text{NR}}{\langle S,U,B\rangle \vdash r \ := e \twoheadrightarrow \ U[v/r]}$$

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

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

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

par
$$\frac{\langle S,U,B\rangle \vdash a_1 \twoheadrightarrow U_1,\ \langle S,U,B\rangle \vdash a_2 \twoheadrightarrow U_2}{\langle S,U,B\rangle \vdash a_1 \ \texttt{|}\ a_2 \twoheadrightarrow U_1 \uplus U_2}$$

seq
$$\frac{\langle S,U,B\rangle \vdash a_1 \twoheadrightarrow U_1; \langle S,U_1,B\rangle \vdash a_2 \twoheadrightarrow U_2}{\langle S,U,B\rangle \vdash a_1 \ \texttt{;}\ a_2 \twoheadrightarrow U_2}$$

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

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

Each action rule produces a list of register updates given an environment $\langle S,U,B\rangle$ where S represents the register state, U is the observable updates, and B represents the local bindings. NR represents the "not-ready" value and can be stored in a binding, but *not* assigned to a register. The strictness of method calls is enforced by checking that parameter values are not NR. Initially U and B are empty and S contains the value of all registers. If the system gets stuck because no rule is applicable, it is assumed than an empty U is returned.

Figure 3-3: Operational Semantics of BTRS Actions

standard aspects of this language involve action composition and the semantics of action guards.

3.3 Action Composition

The language provides 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

Expression Rules:

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

Each expression rule is evaluated in an given an environment $\langle S,U,B\rangle$ where S represents the register state, U is the observable updates, and B represents the local bindings. NR represents the "not-ready" value and can be stored in a binding, but *not* assigned to a register. The strictness of method calls is enforced by checking that parameter values are not NR. One can think of $++$ as list concatenation.

Figure 3-4: Operational Semantics of BTRS Expressions

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 subactions 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.

Sequential composition is more in line with other languages with atomic actions. The

Merge Functions:

$$U_1 \uplus U_2 = \text{error if } \exists r.\{r \mapsto v_1\} \in U_1 \wedge \{r \mapsto v_2\} \in U_2 \quad \text{otherwise } U_1 \cup U_2$$

$$\{\}(x) = \bot$$

$$S[v/t](x) = v \text{ if } t = x \quad \text{otherwise } S(x)$$

When composed, Actions and Expressions are merged using these rules

Figure 3-5: Action and Expression Merging

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.

3.4 Conditional versus Guarded Actions

BTRS has both conditional actions (`if`s) as well as guarded actions (`when`s). 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 an `if`'s predicate evaluates to false, then that action doesn't happen (produces no updates). If a `when`'s predicate is false, the subaction (and as a result the whole atomic action that contains it) is invalid. One of the best ways to understand the differences between `when`s and `if`s is to examine the axioms in Figure 3-6.

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`s in a rule can be treated as an `if` and vice versa.

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

Figure 3-6: When-Related Axioms on Actions

3.4.1 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 3-3, and 3-4. 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).

3.4.2 Derived Rules

The final aspect of BTRS is the notion of Derived rules. In short, derived rules are new rules created by combining primary rules using the combinators listed in Figure 3-7.

Compose Combinator: The compose rule combinator takes two rules as input arguments and produces a rule which behaves like the sequential composition of the two. The seman-

$$\begin{array}{ll} DR ::= & R \\ \quad \| & compose(DR, DR) \\ \quad \| & par(DR, DR) \\ \quad \| & restrict(DR, DR) \end{array}$$

Figure 3-7: BTRS Scheduling Language

tics of the new rule are:

```
compose(Rule A a1 when p1, Rule B a2 when p2) =
          Rule AB (a1 when p1);(a2 when p2)
```

The presence of a guard following a sequential connective raises some important questions, which will be addressed in a following section discussion the lifting of `when` guards. The notable issue, though is that the successful firing of rule AB is only assured when both p1 and p2 are true, but p1 must be evaluated with the effects of a1 visible. Because of the sequential semantics of the primary specification, this new rule is guaranteed to be correct, under the execution semantics described previously.

Par Combinator: The par rule combinator takes two rules as input and produces a rule which behaves like the parallel composition of the two rules. As opposed to the Compose combinator, we now must ensure the mutual exclusivity of the two sub-rules since arbitrary parallel composition could introduce new behavior. We are left with two choices. On one hand we could enforce mutual exclusivity, lifting the guards and taking their union as demonstrated here:

```
Rule AorB (if p1 then a1)|(if p2 then a2)
            when (p1 ∨ p2)
```

On the other hand to guarantee correctness of this combinator on arbitrary rules, we could apply the combinator with the following semantics, guaranteeing that in our new rule we only ever enable one of the rules:

```
par(Rule A a1 when p1, Rule B a2 when p2) =
    Rule AB (if p1 then a1)|(if p2 then a2)
                 when (p1 ⊕ p2)
```

Restrict Combinator: Sometimes we want to express some sort of mutual exclusion, with particular priority. This last combinator does just that:

```
restrict(Rule A a1 when p1, Rule B a2 when p2)=
        Rule AB a2 when (¬p1 ∧ p2)
```

Apart from the semantic richness these rule combinators add, there are advantages in combining rules, since the compiler can do inter as well as intra rule optimization.

Chapter 4

Compilation Scheme

4.1 Syntax Directed Compilation

Each BTRS module is compiled into a C++ class and its rules and methods into class methods. Calling a rule or method on a module instance performs that method on the state and returns the corresponding result. The compilation of actions relies on building shadow registers (in general, shadow modules) to hold speculative state. If the rule fails, these shadows are discarded. To keep track of shadows, the "original" state is recorded well as the most current active shadow for each register. Initially the state map points to the physical values for each register and the shadow map is empty. State maps are consulted to determine the most recent shadow to use for a register read. To perform a register write, we make a shadow if one does not already exist. This shadow analysis is completely static and there is no run-time overhead involved. We use throw and try-catch mechanisms to handle guard failures. The syntax-directed compilation is illustrated using the following example BTRS rule which operates on three `int` registers: `r1`,`r2`, and `r3`:

```
rule R1
((r1 := r1 + 1 when p1) | (r2 := r1 + 2 when p2))
; (if c then (r3 := r1 + 1 when p3))
```

This rule translates to the following procedure:

```
01 void RL_R1(){
02  try{
03    if (!p1){throw guard_failure;}
04    Reg<int> r1_shadow(r1);
05    r1_shadow.write(r1.read() + 1);
06    if (!p2){throw guard_failure;}
07    Reg<int> r2_shadow(r2);
08    r2_shadow.write(r1.read() + 2);
09    Reg<int> r3_shadow(r3);
10    if(c){
11      if (!p3){throw guard_failure;}
12     r3_shadow.write(r1_shadow.read() + 1);
13    }
14    // Commit shadows
15    r1.seqMerge(r1_shadow);
16    r2.seqMerge(r2_shadow);
17    r3.seqMerge(r3_shadow);
18  } catch(int i);
19 }
```

Lines 3-5 correspond to the first action where the effect of throw is to jump out of the try block and therefore the rule. Similarly, lines 6-8 correspond to the second action. Notice that in the third action, the throw happens only when the condition `c` is true and the guard `p3` is false. Also notice that in order to keep all the shadows in scope for the final commit, the shadow for `r3` is built before entering the conditional statement. All three actions read `r1`, but the first two read the original value while the third one reads the shadow value as dictated by the semantics of parallel and sequential action composition. The sequential merges in lines 15-17 commit the updates to the shadows into the original register state.

This example also suggests that each C++ class corresponding to a BTRS module will need a copy constructor to make shadows, and two merge operators `parMerge` and `seqMerge` to merge different shadows of the module in parallel or sequence respectively. The following example illustrates how the parallel merge `parMerge` is used:

```
rule R1
((mod.actionMethodA() when p1) |
  (mod.actionMethodB()when p2))
```

```
void RL_R1(){
 try{
  if (!p1){throw guard_failure;}
   Mod<> mod_shadowA(mod);
   if (!p2){throw guard_failure;}
   Mod<> mod_shadowB(mod);
   mod_shadowA.actionMethodA();
   mod_shadowB.actionMethodB();
   mod.parMerge(mod_shadowA);
   mod.parMerge(mod_shadowB);
 } catch(int i);
}
```

The difference between the parallel and sequential merges lies in how the `modified` bits on the fundamental state elements (those which implement user specified state) are handled. Currently, the only primitive modules we implement that hold user state are a few different varieties of FIFOS, Registers, and Register Files. In all non-primitive modules, the implementation of both parMerge and seqMerge is merely to invoke the corresponding merge routines recursively on all sub modules. In primitive modules, the parallel merge operation unions the final `modified` bits with that of the copy being merged in, while the sequential version always overwrites them. This provides a mechanism which can be used to detect conflicts.

The final output of the BTRS compiler is a collection of C++ classes which each implement the rules and methods specified in the design, a copy constructor, a parallel merge, and a sequential merge routine. In concert with a top-level driver loop which instantiates the objects and drives the rule execution as specified by a particular schedule, we can efficiently implement any BTRS program.

4.1.1 The Details

The Syntax Directed Compilation of the BTRS actions and expressions into C++ is expressed as Haskell pseudo-code in Figure 4-1. BTRS statements translate directly into C statements, while BTRS expressions produce a tuple consisting of a list of C statements and a C expression. The C statements need to be evaluated before the expression. As an example, consider the following expression written as stylized compiler IR:

```
OLet [a = mod1.meth()] in (f(a,a+1))
```

which would return the tuple consisting of a list of (one) statements and the expression corresponding to the value of the let expression:

```
([CStmt (T a = mod1.meth())], f(a,a+1))
```

The translation procedures for both expressions and statements take (as arguments) additional statements which will be evaluated before the statements generated by the *current* object. The statements and expressions produced by the procedures (prefixed with "C") roughly correspond to the C programming language and are trivially converted to their textual representations. The procedures in Figure 4-1 invoke getActiveState and readState to manipulate and read the visible shadow (active) state. These are defined in Figure 4-2.

In addition to the compilation of expressions and actions, the program must have structure. Modules are translated into C++ classes, and schedules synthesized to driver routines. The specified schedule is implemented as a rule in the top-level module. The syntax directed compilation strategy for these elements is given in Figure 4-2. Unlike most transactional systems we have significant freedom in choosing the order of rule execution. The scheduling decisions may have a significant impact on the locality and parallelism exploited in execution.

4.1.2 The Cost of Laziness

It is well known that non-strictness has a cost associated with it. In the case of BTRS, this might have meant that every expression would have returned a predicated value, *i.e.*,the

```
genA :: State -> BTRSAction -> (CStmt, State)
genA s [[r := e]] = (se ++ [gen] ++ [mod.write(ce)], s1)
     where (gen, mod, s1) = getActiveState(s,r)
           (se , ce)      = genE s1 e
genA s [[m.g(e)]] = (se ++ [gen] ++ [mod.g(ce)], s1)
     where (se,  ce)      = genE s et
           (gen, mod, s1) = getActiveState(s,m)
genA s [[if e then a]] = (se ++ gens ++ [if(ce) { ca } ], s2)
     where (se, ce)  = genE s e
           (gens,s1) =
              foldl ($\lambda$ (g,s) mname. (g++g1,s1) where (g1,s1) = getActiveState s) (writtenModules
           (ca, s2) = genA s1 a
genA s [[a when e]] = (se ++ [if(!ce) {throw GuardFail;} ca;], s1)
     where (se, ce) = genE s e
           (ca, s1) = genA s a
genA s [[t = et in a]] = (st ++ [t = ct;] ++ ca, s1)
      where (st, ct) = genE s et
            (ca, s1) = genA s a
genA s [[a1;a2]] = (ca1++ca2, s2)
     where (ca1, s1) = genA s  a1
           (ca2, s2) = genA s1 a2
genA s [[a1|a2]] = (ca1 ++ ca2 ++ merges, s1)
     where news = State {initMap=(activeMap s)+(initMap s),activeMap=emptyMap}
           (ca1, s1) = genA news a1
           (ca2, s2) = genA news a2
           merge modvar = case (s1[modvar], s2[modvar]) of
                              (Just x, Just  y)  -> (x.ParMerge(*y);, Just  x)
                              (Just x, Nothing)  -> ([]             , Just  x)
                              (Nothing, Just y)  -> ([]             , Just  y)
                              (Nothing, Nothing) -> ([]             , Nothing)
           (merges, am) = unzip (map merge subModules)
           s1 = State {initMap = initMap(s), activeMap = am}

genE :: State -> BTRSExpr -> (CStmt, CExpr) -- The CStmts must be evaluated before the CExpr
genE s [[c]]           = ([ ], translateConst[[c]])
genE s [[t]]           = ([ ], t)
genE s [[e1 op e2]]    = (s1 ++ s2, c1 (translateOp op) c2)
   where (s1, c1) = genE s e1
         (s2, c2) = genE s e2
genE s [[ep ? et : ef]] = (sp ++ st ++ sf, cp ? ct : cf)
   where (sp, cp) = genE s ep
         (st, ct) = genE s et
         (sf, cf) = genE s ef
genE s [[e when ew]]    = (se ++ sw ++ [if (!cw) throw GuardFail;], ce)
   where (se, ce) = genE s [[e ]]
         (sw, cw) = genE s [[ew]]
genE s [[t = et in eb]] = (st ++ [t = ct;] ++ sb, cb)
   where (st, ct) = genE s [[et]]
         (sb, cb) = genE s [[eb]]
genE s [[r]]           = ([], getReadState(s,r).read())
genE s [[m.f(e)]]      = (se, getReadState(s,r).f(ce))
   where (se, ce) = genE s [[e]]
```

Figure 4-1: Syntax-Directed Translation of BTRS Actions and Expressions into C++

Maybe type in Haskell. Removing this overhead would result in strict semantics, which are in conflict with the BTRS definition. Alternately, this problem can be fixed by statically lifting `whens` out of let-bound expressions and value methods and remembering their

```
genDefaultDriver modDef = void main { Module top = modDef(); // Construct module
                                      while(1){top.RunRules();}}\\
genModuleDef :: BTRSModuleDef -> C\_ClassDef
genModuleDef [[ModDef name submodules rules meths]] =
class name{genSubmodPtrs submodules
 public:
   genConstructors submodules
   genRule rules
   genMethDef meths
   void ParMerge(Module& x);{foreach submod. submod.ParMerge(*x.submod);}
   void SeqMerge(Module& x);{foreach submod. submod.SeqMerge(*x.submod); }
   void RunRules(){ foreach rule. rule();} };

inits = State {initMap = makeMap state, activeMap = emptyMap}

commitState s = (merges, s1)
  where merges = map ($\lambda$ mod. (initMap s)[mod].SeqMerge(*mod);) (activeMap s)
        s1     = s {activeMap = emptyMap}

getActiveState(s, modname) | (modname $\rightarrow$ v) $\in$ (activeMap s) = ([], v, map)
getActiveState(s, modname) | otherwise                           = (gen, v, s1)
     where gen = [v = ModCopy(initMap(s)[modname]);]
           s1  = s {activeMap = (activeMap s) + {modname $\rightarrow$ v}}

getReadState(s, modname) | (modname $\rightarrow$ v) $\in$ (activeMap s) = v
getReadState(s, modname) | (modname $\rightarrow$ v) $\in$ (initMap s)   = v
getReadState(s, modname) = error ("Nonexistant Module" ++ (show modname))

genRule :: BTRSRule -> ClassMemberFunction
genRule[[Rule name a]] = void name(){ try{ ca; commitState(s)} catch(int i);}
     where (ca, s) = genA inits a

genMethDef :: BTRSMethodDef -> C_MemberFunction
genMethDef [[ActMeth $name \lambda x.a$]] = void name(argT x){ca; commitState(s)}
     where argT    = typeInfo g
           (ca, $\rho$) = genA inits a
genMethDef [[ValMeth $name \lambda x.e$]] = retT name(argT x){se; return ce;}
     where (se, ce)    = genE inits e
           (retT,argT) = typeInfo f

class Reg {
  bool modified;
  int  state;
public:
  Reg(){ modified = false; state    = 0; // init value }
  Reg(Reg& x){modified = false; state = x.state} // make shadow
  int read(){ return state;};
  int write(int x){state = x; modified = true;};
  void ParMerge(Reg& r){ state = *r.read(); modified |= true;
       if(modified && r -> modified) {throw Error;}} // double Par write
  void ModMerge(Reg& r){ state = *r.read(); modified |= *r.modifed; }};
```

Figure 4-2: Syntax-Directed Translation of BTRS Module Definitions, Helper Functions, Top-level Driver, and Register Class Definition

original position with respect to any conditional code. If conditional use is not exercised in a particular firing, then the guard failure exception is not thrown, resulting in lazy let semantics. This lifting is accomplished during one of the compiler optimization phases.

4.2 Compiler Phases

The BTRS compiler employs a multi-phase approach in performing the compilation of a program specification. These phases are depicted in Figure 4-3 and discussed in the following sections. The top-level compiler function (main) is in CompileATS.hs and all the compiler phases are invoked in the function "compile" in that file.

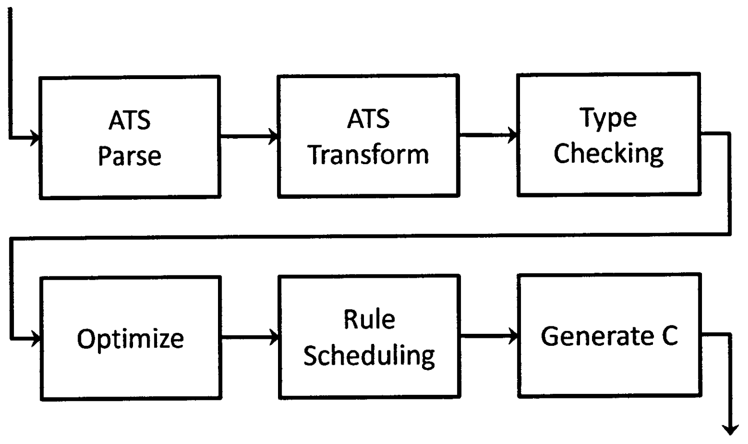

Figure 4-3: BTRS Compiler

4.2.1 Internal Representation

The program structure is represented as an array of Module Definitions, defined by the structure ModuleDef shown below:

```
data ModuleDef =
  ModuleDef {
    modDefAttrib      ::  Attrib,
    modDefName        ::  ModuleDefName,
    modDefArgs        ::  [BindName],
    modDefLocalDefs   ::  BindList,
    modDefSubMods     ::  [ModuleInst],
    modDefRules       ::  [Object],
    modDefMethods     ::  [MethodDef],
    modDefIsNameSpace::  Bool
   }
 deriving(Eq, Show, Data, Typeable)
```

Each ModuleDef's type is its name, and contains a list of sub modules which takes the form of a list of tuples pairing scoped identifiers with other ModuleDefNames. With a programmer-specified root module, there is enough information to construct a complete module hierarchy. The other fields are relatively self explanatory: modLocalDefs binds names which are used throughout the module definitions rules (modDefRules) and methods (modDefMethods).

A recurrent theme throughout the IR data structures is the inheritance of Haskell type classes Data and Typeable. The Data typeclass exposes a canonical representation of an algebraic datatype's structure. The Typeable typeclass exposes type information, allowing the Haskell compiler to determine the type of an object by inspection. Together these two typeclasses allow for very powerful yet concise representations of compiler transformations such as the example below:

```
liftParWhens_Obj ::  Object -> Object
liftParWhens_Obj o = ...

liftParWhens ::  (Data a) => a -> a
liftParWhens = everywhere (mkT liftParWhens_Obj)
```

The operation could be interpreted as: "traverse the entire program structure, and every time you find an object of type tt, transform it using the function trans, which is of type tt → tt". This style of generic programming is a very powerful feature of the Haskell programming language.

While ModuleDefs are the building blocks of the module hierarchy and encapsulate all intermodule communication, the actual computation contained in the rules and methods is represented using the [recursive] algebraic datatype Object, shown below:

```
data Object
  = OSeq          Attrib [Object]
  | OPar          Attrib [Object]
  | ORestrict     Attrib [Object] Object
  | OLit          Attrib Value
  | OPrim         Attrib PrimName
  | OBoundVar     Attrib BindName
  | OMethCall     Attrib MethodName
  | OWhen         Attrib Object Object
  | OIf           Attrib Object Object Object
  | OWhile        Attrib Object Object
  | OWhileGuard   Attrib Object
  | OLet          Attrib BindList Object
  | OApply        Attrib [Object]
  | ORule         Attrib RuleName Object
  | OLocalGuard   Attrib Object
  deriving(Eq, Show, Ord, Data, Typeable)
```

OSeq and OPar are the sequential and parallel composition of actions. ORestrict restricts the execution of its third parameter (presumably some action) by the successful execution of its second parameter (a list of actions). This is very close to the semantics of the sequential composition. Following that are literals, primitives, bound variables, and method calls. OWhen guards the second argument with the return value of the third (a Boolean value). OLocalGuard is similar to OWhen, except that it provides a barrier for guard-lifting (described in a following section). OWhile and OWhileGuard are both looping structures, except one relies on a guard failure to exit the loop while the other has an explicit loop condition. OApply is used to apply methods (functions etc.) to arguments. The semantics of ORule and OLocalGuard are identical, providing a mechanism to shield the scope of a guard failure. In the IR, all datatypes have an Attribute field, which stores (among other things) the type of a particular instantiation.

The IR is sufficiently simple (effectively annotated BTRS) that the optimization phases can be viewed as source to source transformations, with the SDC occurring in the final stage of C generation. All the arithmetic data types for the IR are defined in Types.hs.

4.2.2 ATS Parse

Parsec, an industrial strength, monadic parser combinator library for Haskell [10] was selected to parse the input language. The Sequential connective is currently introduced through a side channel in which a schedule (or various scheduling constraints) are supplied to the compiler. The parse phase constructs a structure that is roughly equivalent to the internal representation (IR), storing all ancillary information in state tables for later use. For reasons discussed in the following chapter, the input language to this compiler is in a form known as ATS. The code for this phase is located in ATSParse.hs and BTRSParse.hs.

4.2.3 ATS Transform

In this phase of the compilation, the output of the Parse phase is transformed into the compiler's IR. Though the same data structures are used between these two phases, there are particular structural invariants which are enforced between all compiler phases to reduce the complexity of the tree traversals. It would have been possible to restrict patterns that are considered illegal through the definition of the data types used to build the IR, but in an attempt to keep the IR data types clean and simple, this approach was chosen. There are some heavy-weight transformations which occur in this phase, since we need to handle some unfortunate features in ATS. This phase should be as a source to source transformation which smooths out some of the idiosyncrasies of the input structure. These are all contained in the function doATSTransform, which is located in the file Transform.hs

A primitive library of polymorphic functions and modules was introduced to ease the task of programming. The first task is to handle the these primitives. This is accomplished by first exposing all primitive functions as global bind-names. Since all types in our IR are monomorphic, these primitives need to be inlined so that type-specialization can occur. After that, there are a number of small fix-ups designed to cannonicalize various logical structures after which let bindings are topologically sorted. Next, all name bindings are localized from the Module context to their particular rule or method. While this does involve the duplication of some bindings, there are performance advantages in having many smaller localized bind-lists over one global module-context namespace. One invariant enforced is that let expressions can contain no unused let bindings.

Another important issue is that of ActionValues, which are part of the ATS syntax but which have been deemed unnecessary in BTRS since they are syntactic sugar for combinations of value and action methods. The following pseudo code illustrates a typical use of an action value:

```
OLet [ an = ...
      ,bn = mod.ActionValue() -- wants just the "value" part
      ,cn = ...]
in
     OPar [ mod.ActionValue() -- wants just the "action" part
           ,mod.Action1()
           ,mod.Action2(bn)]
```

It is possible to decompose the ActionValue into its action and value components, however a substantial increase in code duplication was observed, especially in guard evaluation. In addition, any side-effects of the action-value might then be observed in a different order due to action reordering resulting from the change in data dependencies. Bluespec Inc. compiles ATS to efficient hardware, so our initial correctness metric was trace equivalence with the Bluesim simulator. The correct ordering dependencies had to be maintained to avoid reordering the sequential interpretation of the parallel action composition contained in OPar. An efficient approach which both avoided code duplication and maintained trace equivalence with Bluesim involves the introduction of a fake use (shown in Figure 4-4). The data-flow edges remain unchanged, prohibiting the binding of *bn* to be moved below the original location of the "action" component of the ActionValue invocation. The result is a single invocation of the ActionValue, which simplifies matters greatly and cuts down significantly on the amount of buffered state.

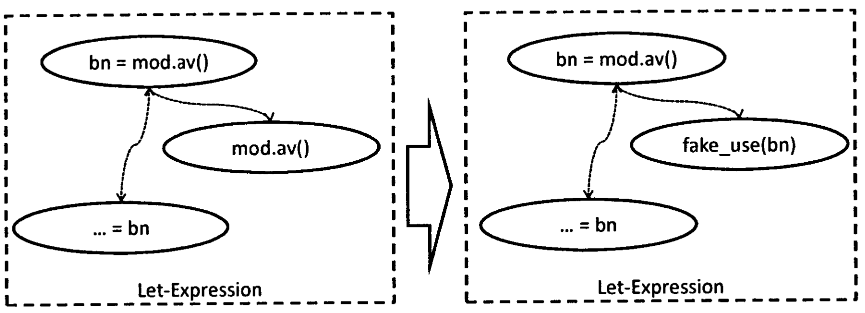

Figure 4-4: Fake use Introduction

Next, all name bindings are pushed to the lowest possible point in the data-flow graph. This has obvious benefits when generating conditional code, and is also one of the IR invariants. More importantly, this phase is used to localize when-guards which is a requirement in supporting lazy let semantics. A final round of small clean-up passes designed to regularize structure completes the conversion of ATS to IR.

4.2.4 Type Checking

Though type complete, the ATS input language is not fully annotated. The presence of primitive functions adds a degree of polymorphism which needs to be resolved. For complete type unification, proviso annotation is also required. Primitive function types are accompanied by provisos, specifying relations between the types. Take, for example the following annotation for the primitive function extract:

```
PrimDef{ pname    = mkBN "extract",
         pstrtype = "forall n m o.  (TSubl 1 2 m) =>
                      Bit n -> Bit o -> Bit o -> Bit m",
         pctrans  = prim_expr_extract}
```

Notice that that the proviso on the type of the primitive extract function indicates that the difference between the first and second bit widths results in the third width. This is what one would expect, given an intuitive understanding of bit extraction. With these annotations on our primitive functions, we are able to fully annotate the types of every object in our module list using the following standard type unification algorithm:

1. Assign a new type variable to each object.
2. Initialize a type environment where each type variable is unknown.
3. Unify types in a top-down traversal, each time you encounter a primitive function or an existing type annotation, enrich the type environment, generating new type variables as needed. Store the provisos for final fix up.
4. At this point, each type variable should map to a concrete type, if they don't, apply the provisos.
5. Assert an error if there are remaining unknown type variables.

4.2.5 Optimization

The Optimization phase contains great potential for increasing the efficiency of generated code. Unfortunately, this thesis is being written before much exploration has been undertaken in this area, but this will be developed further as we refine our compilation techniques. Currently, two major optimizations (seqPars, and liftParWhens) are undertaken along with some minor fix ups.

The optimization seqPars finds actions composed in parallel and, sequentializes them under certain conditions. To motivate this optimization, consider the following IR snippet, which sequentially composes three actions on the same module:

```
OSeq [ mod.actionMethodA()
     , mod.actionMethodB()
     , mod.actionMethodC() ]
```

As specified by our syntax directed compilation scheme, this would generate the following C code:

```
Mod<> mod_shadowA(mod);
mod_shadowA.actionMethodA();
mod.seqMerge(mod_shadowA);
Mod<> mod_shadowB(mod);
mod_shadowA.actionMethodB();
mod.seqMerge(mod_shadowB);
Mod<> mod_shadowA(mod);
mod_shadowA.actionMethodC();
mod.seqMerge(mod_shadowC);
```

Consider the semantics of the Sequential connective (described in Section 3.2) which stipulate that a sequential composition succeeds **iff** all the composed actions in the sequence successfully execute. Since a method failure throws an exception, we can optimize the previous sequence as shown below with the understanding that the final state commit will be bypassed if any of the actions fail. Sequential compositions require a constant sized shadow state, whereas the amount of shadow state for parallel compositions in creases as a function of the number of actions being composed.

```
Mod<> mod_shadow(mod);
mod_shadow.actionMethodA();
mod_shadow.actionMethodB();
mod_shadow.actionMethodC();
mod.seqMerge(mod_shadow);
```

Since a sequential implementation of BTRS can execute sequences more efficiently than parallel compositions, removing parallel compositions has significant performance benefits as there is a large overhead to simulating them on a sequential machine. To demonstrate the process of sequentializing parallel compositions, consider the structure where two actions are composed in parallel:

```
OPar [ mod.actionMethodA()
     , mod.actionMethodB()]
```

where actionMethodA, actionMethodB, and actionMethodC are defined as as follows:

```
Module mkMod(ModIFC);
  Reg(t) a <- mkReg();
  Reg(t) b <- mkReg();
  Reg(t) c <- mkReg();
  ...
  method Action actionMethodA();
    a <= c+1;
  endmethod
  method Action actionMethodB();
    b <= a;
  endmethod
  method Action actionMethodC();
    c <= a+1;
  endmethod
endmodule
```

Efficient shadow management is of greatest importance to the performance of the generated code. With no optimization, the implementation of this composition would require mod to be shadowed three times. A quick analysis of the implementations of the actionMethods shows that a sequential composition results in the same state change as their parallel composition, but requiring only half the shadow state. Notice, though, that there is no sequential interpretation for the parallel composition of actionMethodA and actionMethodC.

The manner in which this optimization is performed is really quite simple. First a graph is created where each node in the graph corresponds to an action in the parallel sequence. An edge from action A to action B indicates state written in B is read in A. Cycles indicate dependencies which are non-sequentializable, and a topological sort will produce the correct order in which to sequentially compose the actions. A further refinement of this procedure breaks non-sequentializable actions into parallel compositions of their sub-actions. Without breaking atomicity or changing the semantics of the original program, this increase in granularity allows for the sequentialization of some of these sub-actions. This technique effectively descends down the module hierarchy until we reach the point where the minimal amount of state needs to be shadowed. This action relies on knowledge of whether an action can fail, or not. This analysis is performed by looking for embedded `when` clauses.

Shadow state serves two purposes: it implements parallel composition, and guarantees atomicity by requiring that an action complete successfully before committing its state. In the absence of potential guard failures, seqPars can remove all non-essential uses of shadow state in a program. The next optimization, liftParWhens, is directly related this point. If we can lift all when clauses out of methods and rules, we can tell before invoking it, whether or not it will fail, and assemble the corresponding action more efficiently. Consider the following snipped of IR:

```
(ORule "r1" (OPar [ (OWhen mod.actionMethodA() guardA)
                  , (OWhen mod.actionMethodB() guardB)]))
```

This rule, consisting of the parallel composition of two guarded actions would produce the following C code:

```
void r1(){
  try{
    if(!guardA) throw guard_failure;
    Mod<> mod_shadowA(mod);
    mod_shadowA.actionMethodA();
    Mod<> mod_shadowB(mod);
    if(!guardB) throw guard_failure;
    mod_shadowB.actionMethodB();
    mod.parMerge(mod_shadowA);
    mod.parMerge(mod_shadowB);
  } catch ( int i);
}
```

There is a large performance overhead in using try/catch to handle potential guard failures. If the guards (both explicit and implicit) could all be lifted to the top-level, we could determine before its execution whether a rule will succeed or not, removing the need for try/catch blocks entirely. It turns out that this will only work for parallel actions since the guard of a sequenced action cannot be lifted. It needs to be evaluated in an environment where the state changes of the previously executed actions are visible. As a result, a companion CAN_FIRE* method is created for each rule and then incorporated into the final scheduling logic. Lifting the parallel guards of the previous example produces the following C code:

```
bool CAN_FIRE_r1(){
  return (guardA && guardB);
}
void r1(){
  Mod<> mod_shadowA(mod);
  mod_shadowA.actionMethodA();
  Mod<> mod_shadowB(mod);
  mod_shadowB.actionMethodB();
  mod.parMerge(mod_shadowA);
  mod.parMerge(mod_shadowB);
}
```

4.2.6 Rule Scheduling

While substantial effort has gone into compiling the rules efficiently, there are limits to the optimization of individual rules that significantly improve the code quality of a sequential implementation. Rule scheduling offers the potential for additional optimization opportunities, but the work in this area is only in the beginning stages. Depending on the target architecture a maximally parallel schedule partitioned to avoid resource conflicts would be preferable, while a single-core machine might work best under a schedule as simple as a brain-dead loop over all the rules.

Suppose the programmer has specified a system in which two rules interact with a memory. rule1 writes consecutive addresses starting at zero, while rule2 reads from consecutive addresses starting at the same point. With no specified rule priority, the wrong scheduling order might result in the reading of uninitialized memory. It is unlikely that the programmer will recognize that they have under-specified the system, because the error is hidden by the schedule. Problems may arise if a new (though still legal) rule schedule is selected, since the design may cease to function as desired. By then, it would be difficult to determine whether the error is in the specification or the compiler. The resolution usually involves removal of scheduling non-determinism. This was the case with the more complex designs which were used to test the compiler, with the unfortunate side effect of removing the possibility to perform any meaningful experiments with different schedules.

To implement all user specified and compiler derived schedules, a number of scheduling combinators were developed whose semantics are a super set of the derived rule combinators described in Section 3.4.2. Further work is required to make the test applications robust enough so that they can be used to evaluate the effect of different derived rules. They are currently used only as a convenient mechanism for expressing general rule schedules. These combinators prove to be quite useful since any schedule can be expressed as a single derived rule, which is generated in the scheduling phase.

4.2.6.1 Scheduling Combinators

Try Combinator: The Try combinator, of type *Rule* $\rightarrow$ *Rule*, creates a derived rule which isolates any guard failures. The following example uses this scheduling combinator to construct a rule which will never fail.

```
try(Rule A a1 when p1) = Rule A'(if p1 then a1) when True
```

Compose Combinator: Suppose we wanted to create a rule which implements the schedule that fires all the rules A..C in a sequence. It should be obvious that the Compose combinator (which is of type *Rule* $\rightarrow$ *Rule* $\rightarrow$ *Rule*) is the right choice. Using the compose combinator in isolation doesn't result in a very useful schedule:

```
Compose(Rule A a when pa,
        Compose(Rule B b when pb,
                Rule C c when pc)) =
                      Rule ABC (a when pa);(b when pb);(c when pc)
```

This has the obvious flaw that if any of the predicates fail, the entire rule fails and no state will ever get committed. Wrapping each primary rule in the `try` combinator will give use a much more reasonable derived rule, where each rule is "tried" in sequence.

Restrict Combinator: Given two rules $rule1$ and $rule2$, this combinator produces a rule whose guard is the union of $rule1$'s guard and the inverted guard of $rule2$. This produces a very straightforward combinator of type *Rule* $\longrightarrow$ *Rule* $\longrightarrow$ *Rule*, which encodes rule priorities.

```
Restrict(Rule A a when pa, Rule B b when pb) = Rule ArB (a when pa ∧ ¬pb)
```

Loop Combinator: This combinator is not part of the BTRS's rule scheduling syntax, but it is thought that it might be useful in the future. It's semantics are to fire repeatedly until the guard fails in this manner:

```
loop(Rule A a when p) = Rule AA (while p a) when True
```

PAR Combinator: As its name suggests, this combinator composes the bodies of mutually exclusive rules as follows:

```
PAR(Rule A a when pa, Rule B b when pb) =
                                      Rule ApB (OPar[a,b] when pa ∧ pb)
```

4.2.6.2 Esposito Schedules

In addition to the derived rules which are part of the BTRS language definition, we support a particular format of schedule annotation which is output by the Bluespec compiler (for reasons discussed in the following chapter) and introduced through a side channel as it is **not** part of the BTRS language. For historical reasons, this is referred to as the *Esposito* schedule.

The Bluespec compiler targets the maximally parallel schedule for which there is a corresponding sequential interpretation. Since it is optimized for hardware generation, the maximal parallelism makes intuitive sense. The motivation behind an insistence on a sequential interpretation is that it makes it possible to reason about the effect of a particular rule in isolation. Without this restriction, rules exhibit different behavior depending on the rule schedule. Lack of sequentializability would also violate the execution semantics of BTRS, as described in 3.1. The example shows how a system of rules and a corresponding Esposito schedule can be converted into an equivalent specification consisting of a single derived rule:

```
rule   blocked-by
----   ----------
a      [b]
b      [c]
c      [d]
d      []

order
-----
[a,b,c,d]
```

which would produce a schedule of the the form:

```
let a´= try(restrict(a,b))
    b´= try(restrict(b,c))
    c´= try(restrict(c,d))
    d´= try(d)
in
    compose(a,(compose b,(compose c,d´)))
```

4.2.6.3 Schedule Optimization

In addition to these combinators, a few attempts at schedule optimization were made, though the effectiveness of these experiments were inconclusive as any improvements were vastly overshadowed by the extremely inefficient nature of our input programs (ATS). One such attempt involved the creation of rule sensitivity lists through static data-flow analysis. The motivation for this was the prohibitively large size of some of the rule guards, resulting from our aggressive guard lifting. The idea behind this optimization is that once a rule's guard has evaluated to false, there is no need to re-evaluate it unless the guard's read-state has been written.

Apart from submitting a complete schedule requiring the use of hierarchical names to refer to rules at various depths in the module hierarchy, the programmer can specify scheduling restrictions on a per-module basis in either Esposito or BTRS formats. These are then applied in a bottom-up manner to the module hierarchy, propagating constraints between modules based on the use of interface methods. This approach can lead to an error condition since it is possible to specify an unimplementable schedule because the user is allowed to explicitly refer to interface methods when giving a sequential interpretation of a module's behavior in isolation. Consider the example where module B is a sub-module of A, each module has rules x and y. Module B has interface method foo, which is invoked by A's rule x, and method bar, invoked by A's rule y. The following ordering constraints are contradictory:

```
Module A:
---------
order [x,y]

Module B:
---------
order [x,bar,y,foo]
```

Obviously, it is impossible to implement this rule schedule, which brings up a grey area in the proposed rule-scheduling approach. In the section on derived rules (3.4.2) we introduced a mechanism for scheduling rules, but not methods. At some level, scheduling methods is unnecessary if the programmer is interested in expressing the restrictions only on the global scale, though often thinking about global schedules is prohibitively complex, making it desirable to reason about local interactions in isolation.

For a sequential implementation, the ultimate goal in the scheduling process is to generate a schedule which has a closer relationship to the implemented algorithm than a simple iteration over all the rules, though it is unclear whether this will have any performance benefit. This is the subject of ongoing investigation.

4.2.7 C Generation

C generation is conceptually one of the simpler aspect of the compilation task, although the implementation is one of the more complex, so as to improve the efficiency of the generated code. The SDC is outlined in Figures 4-2, and 4-1. The function generateC is overloaded, so for all constructs requiring shadowed state, the function guaranteeShadowState is invoked to make sure that all modified state has been duplicated, and that the shadows are mapped to the correct names. This amounts to keeping a stack for each variable name on which to push and pop the shadows as they are created and merged.

4.2.7.1 C Library Implementation

A substantial library of primitive functions and modules is assumed and must be implemented in order to execute programs, consisting of the usual suspects handling all logical and bit-wise operations, as well as some more exotic data formatting and printing utilities. Having the largest effect on the performance of the generated code is the implementation of the library modules, foremost among which were the Wide data type, Registers, Register Files, and FIFOS. Initial implementations showed excessive copying during shadow creation and merging. To mitigate the overhead of shadowing, lazy primitive modules which copied data only "on demand" were implemented.

The impact of lazy library modules depends on the ability of the optimization seqPars to minimize the amount of shadowed state. A lazy module copies data only on demand, which usually means that it is being modified. Consider the case of a register file: a register file being used concurrently in parallel would need to have two shadows created. Once modified, the parallel merge operation is then used to commit the state while checking for conflicts. Keeping non-committed updates in a linked-list structure and referring all reads back to the original module allows shadow creation to avoid copying any of the user-visible state. Because each lazy module has a pointer back to the state it is shadowing, a reader can always access the most recently written version. Similar approaches were taken for the registers and FIFOS.

Chapter 5

Input Language Details

With no special interest in designing yet another input language, the decision was made to hook into Bluespec Inc's BSV compiler (BSC). While this approach has some significant problems, there are equally significant benefits in being able to leverage the substantial capabilities of BSC's front-end and all the designs which are currently implemented in BSV (an important point when trying to exercise a compiler). The liabilities in using BSC as the front-end stem from the fact that BSC is a closed-source proprietary compiler. Since there is no access to the source code, debugging information dumped between compilation phases is used to reverse-engineer their internal representation.

As shown in Figure 5-1, the front end consists of two phases, type-checking (resolution) and static elaboration. After the first phase, the BSV program is in a form known as I-Syntax, which amounts to a fully-typed λ-calculus. The static elaboration phase leaves the program in a form known as ATS. ATS is fully typed and fully elaborated, consisting of a module hierarchy, rule and method definitions, and local bindings. In this form, it is very close to BTRS, as described in Section 3.2.

We have the option of using I-Syntax as the input language, though to do so would require us to implement our own static elaborator which is a prohibitive amount of work. It would seem then that using ATS is the natural choice except for the fact that all of the data-type information and much of the program structure (information which could help us

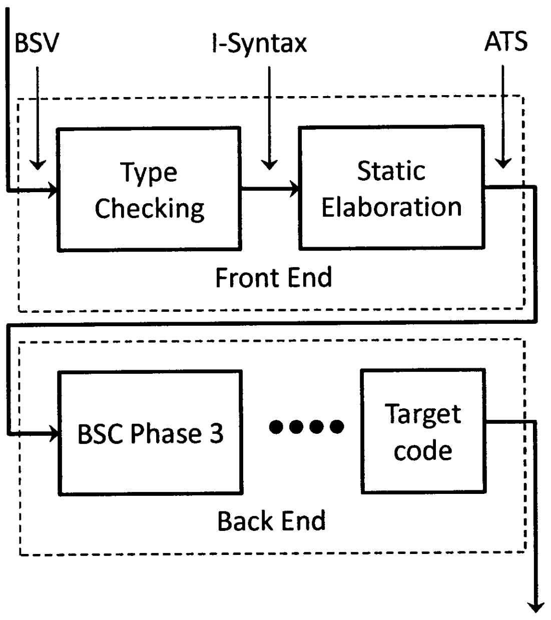

Figure 5-1: Simplified view of BSC

generate more efficient software) has been removed. The only data types in ATS are bit vectors.

Choosing BSC's ATS over I-Syntax as our input language meant choosing type and data-flow re-construction over re-implementing the static elaborator. Since BSC is targeted for efficient hardware generation, lots of valuable program structure is removed by the front-end. In general, the problem of static elaboration is quite difficult. There are certain types of structures which are preferably elaborated by BSC, which we classify as structural elaboration. An example of this is shown below where the loop is instantiating interface methods (the library function fifoToPut will itself be inlined):

```
// Type Declaration:
interface Vector#(3, Put#(type t))) in;
// Implementation:
// instantiate fifos
Vector#(3, FIFO#(t)) in_fifos <- replM(mkFIFO());
```

```
// implement the interface methods
in = map(fifoToPut, in_fifos)
```

becomes:

```
FIFO#(t) in_fifos_0 <- mkFIFO();
FIFO#(t) in_fifos_1 <- mkFIFO();
FIFO#(t) in_fifos_2 <- mkFIFO();
interface method in_0 = fifoToPut(in_fifos_0);
interface method in_1 = fifoToPut(in_fifos_1);
interface method in_2 = fifoToPut(in_fifos_2);
```

Other static elaboration corresponds to the actual data-flow description, such as the example shown here:

```
// Type Declaration:
Vector#(3, Reg#(Bit(n))) ba <- replM(mkReg());;
...
// Loop:
Bit#(n) lv = 0;
for(Int i = 0; i <= 3; i=i+1)
  begin
   lv = lv + ba[i]
  end
```

becomes:

```
Reg#(t) ba_0 <- mkReg();
Reg#(t) ba_1 <- mkReg();
Reg#(t) ba_2 <- mkReg();
...
Bit#(n) lv_0 = 0    + ba_0.read()
Bit#(n) lv_1 = lv_0 + ba_1.read()
```

```
Bit#(n) lv_2 = lv_0 + ba_2.read()
```

Obviously, this kind of structure can be very efficiently expressed in software and its removal is unfortunate.

Lastly, there are certain library primitives such as dynamically-indexable arrays which are unfortunately inlined since they have quite natural and efficient implementations in SW. The example below shows this behavior in which BSC turns array indexing into a cascaded *if* statement, which is a natural implementation in hardware, but is horribly inefficient in software.

```
// Type Declaration:
Vector#(3, Reg#(Bit(n))) a <- replM(mkReg());;
...
// some function:
...
return a[x];
```

becomes:

```
Reg#(t) a_0 <- mkReg();
Reg#(t) a_1 <- mkReg();
Reg#(t) a_2 <- mkReg();
...
return (if (x==0) return a_0
        else if(x==1) return a_1
        else if(x==2) return a_3)
```

Similar bit blasting occurs when using Structs, turning what could be a simple member selection into a complicated (and very expensive) bit-extraction procedure. Of course, it is possible to reconstruct the Data-flow structures, data types, and some of the primitive functions through the use of pattern-matching and data-flow analysis. Work in this area is ongoing, though it is not reflected in the performance numbers reported in this thesis.

Perhaps the biggest problem of all, is that significant work would be required to add the Sequential Connective to the BSV language. We currently use an ad-hoc solution which introduces this operator through a side channel. How we eventually solve this problem is the subject of ongoing negotiations with Bluespec Inc.

Chapter 6

Evaluation

To evaluate the approach, a BSV specification of the H.264 video decoder was. Some time will be spent describing the application in order to more clearly discuss the hazards associated with its performance. This particular application was chosen since it appears to be a good candidate for eventual HW/SW synthesis. Some parts of the computation (described in the next section) seem well suited for software while others would benefit from hardware acceleration.

6.1 H.264

The H.264 Advanced Video CODEC is an ITU standard for encoding and decoding video with a target coding efficiency twice that of H.263 and with comparable quality to H.262 (MPEG2) [8, 13]. H.264 enables PAL (720×576) resolution video to be transmitted at 1Mbit/sec. The BSV specification used in this evaluation was written by Chun-Chieh Lin, with the ultimate goal of ASIC implementation. The ASIC target greatly affected the decoder architecture (taking on a very different character that an FPGA or Software implementation might have), and is documented in his Masters Thesis [11].

The computational requirements of decoding H.264 video vary depending on video resolution, frame rate, and level of compression used. At the low end, mobile phone appli-

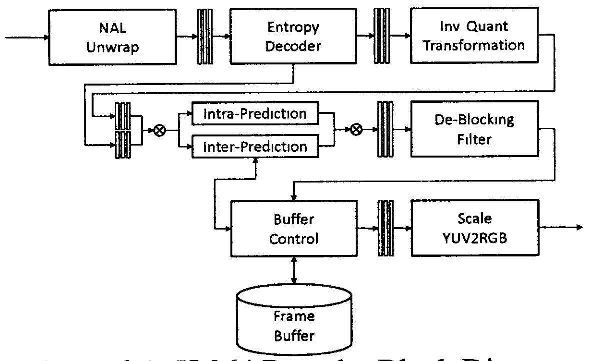

Figure 6-1: H.264 Decoder Block Diagram

cations favor videos encoded in the QCIF format (176 × 144) at 15 frames per second. At the high end of the spectrum, HD-DVD videos are encoded at 1080p (1920 × 1080) at 60 frames per second.

H.264 reconstructs video at the granularity of 16 × 16 pixel macroblocks, which may be further subdivided in some decoding steps. H.264 uses two main techniques to reduce the number of bits necessary to encode video. Intraprediction infers macroblocks in a frame from other previously-decoded spatially-local macroblocks in the same frame. Interprediction infers macroblocks from indexed macroblocks in previously decoded frames. Figure 6-1 shows a block diagram of the H.264 decoder.

NAL Unwrap: The Network Adaptation Layer (NAL) interprets sequences of bits and marks the stream with the coarse grain packeting information, effectively working as a stream parser. The NAL also extracts high-level control information and passes it downstream to subsequent blocks. This stage is not computationally intensive and could easily end up as software.

Entropy Decoder: The H.264 CODEC uses variable-length entropy coding to encode integers. Two techniques are used to accomplish this: CAVLC(Context Adaptive Variable Length Coding) and CABAC(Context Adaptive Binary Arithmetic Coding). Both techniques feature context-aware bit-mappings that vary during decoding. This step is accomplished by setting up look-up tables in memory, and directed by control bits parsed by the NAL Unwrap phase.

Inverse Transformation and Quantization: H.264, like many video CODECs, represents data via a fixed prediction, based on previously decoded image data coupled with a residual error value representing the difference between the fixed prediction and the original image. A lossy, low-pass discrete cosine transformation is employed to develop a compact representation of the residual values. H.264 also allows variable quantization of DCT coefficients to enhance coding density. DCT's parallelize well and are therefore well suited to hardware.

Intraprediction: Video frames have a high amount of spatial similarity. Intraprediction use previously decoded, spatially-local macroblocks to predict the next macroblock. Intraprediction works well for low-detail images.

Interprediction: In video, temporally local frames often exhibit only small differences. Interprediction attempts to capitalize on this similarity by encoding macroblocks in the current frame using a reference to a macroblock in a previous frame and a vector representing the movement that macroblock took to a $\frac{1}{4}$ pixel granularity. The decoder uses an interpolation process known as *motion compensation* to generate the prediction value. This phase is computationally intensive, requiring massively parallel computation as well as substantial memory bandwidth.

Deblocking Filter: Since lossy compression used to encode pixel blocks in H.264, decoding errors appear most visibly at the block boundaries. To remove these visual artifacts, the H.264 CODEC incorporates a smoothing filter into its encoding loop. However, not all inter-block discontinuities are undesirable; edges in the original image may naturally occur on block boundaries. H.264 incorporates fine-grained filter control to preserve these edges.

Buffer Control: H.264 does not require interpredicted images to depend on temporally-local, temporally-ordered images. Rather, frames can be predicted from previously decoded frames corresponding to frames far in the past or future of the video. Buffer control maintains a set of previously decoded frames and is responsible for handling the in-stream requests to access (*e.g.,*delete, prediction logic reads, writes from deblocking) these frames in its store.

We note in passing that H.264 decoding entails a large amount of computation (as many as 30 8-bit or 16-bit fixed-point multiplies per pixel). Most of these computations take place in four blocks – Inverse Quantization, Inter- and Intra- prediction and the Deblocking filter. In addition, it involves the movement of large amounts of data, the implications of which are discussed below.

6.2 Performance/Correctness

The application is correctly compiled, though it leaves much to be desired in way of performance. The complexity of this application required considerable time and effort (about two man-years) to get the compiler to sufficient maturity. Our benchmark infrastructure used a bit-wise comparison to ensure the correctness of our decoded video against that of the reference decoder. As it currently stands, the produced code is about 300x slower than the hand-coded C reference implementation.

6.2.1 Performance Analysis

Central to the hardware-inspired methodology is the notion of moving the data through the algorithm, rather than moving the algorithm over the data. The former is well suited for hardware generation, while in software passing a pointer is a far more efficient method of communication, especially if the message is quite large. A sensible compiler optimization would try to infer module structure such that we could replace entire modules with a more SW friendly implementation. Take for example, the following snippet of a direct translation of a buffered FIFO with the interface methods enq and deq, and first:

```
template<typename T> class SizedFIFO{
  inline T first(){
    return arr[deqIdx];
  }
  inline void enq(T x){
    modified_enq  = true;
    arr[enqIdx] = x;
    (++enqIdx) %= maxCount+1;
  }
  inline void deq(){
    modified_deq = true;
    (++deqIdx) %= maxCount+1;
  }
}
```

Fifos like this are primarily used to pass data between modules and are used to connect the pipeline stages of the H.264 design. Notice that the interface methods first and enq copy the data. If this process is parametrized with wide datatypes, a common occurrence in the designs we have seen, a simple data-flow token transfer can incur a serious memory access overhead. An implementation using pointers would be far more efficient, though it is unclear the extent to which we could perform the necessary structural modifications to the module hierarchy. So far, effort has been spent trying to optimize the code as it has been written, but it is this kind of optimization which would allow us to replace a literal translation of a HW FIFO (as shown above) with a more efficient SW version that will allow us to compete with hand-written code.

The H.264 program specification we compiled consists of roughly eight stages, each of which is connected by a FIFO similar to the one previously described. The memory containing the video stream is copied at least eight times while being decoded. Additionally, within each pipeline stage, the data is copied multiple times, since we are effectively simulating a hardware design using software. This inherently hardware-centric specification, combined with the deficiencies of ATS discussed in Section **??** explain the relatively poor performance.

Most of the pipeline stages of the H.264 decoder are implemented in a distributed (maximally parallel) and latency-insensitive manner. This implies that the modules themselves have ample internal buffering (generally at least one frame's worth). Within the modules implementing each pipeline stage, multiple threads of execution (rules) work on the data in parallel. In hardware, this is ideal, though in software, it implies extensive copying of the data as it is taken from the frame-buffer, to the registers or local memory used by an individual rule. For example, in Intraprediction, there are a number of parallel work units, each of which processes a single macro block. These blocks are copied multiple times before their processing is complete. The question central to this type of problem is whether the Compiler should be able to convert these types of control structure to equivalent SW-friendly forms, or if the programmer should specify the system differently depending on the target (HW or SW).

Chapter 7

Related Work

This chapter is not intended to be a comprehensive review of all parallel languages. Instead, the defining features of BTRS are identified and examples of other languages which share these features are given. The novelty of BTRS semantics lie in the combination of guarded atomic actions combined with user-level scheduling for software synthesis. Our scheme does not require the programmer to fully specify a schedule, but instead allows the user to specify particular scheduling restrictions only where necessary. The programmer has the freedom to under-specify the system, giving the compiler a larger space of rule schedules to explore. The result is four very powerful language features:

1. **Explicit Communication**: Since we have no pointers or reference types, problems associated with aliasing are completely avoided, easing the task of optimizing in multi-threaded contexts.

2. **Non-determinism**: Leaving non-determinism in the specification gives the compiler greater range in searching for efficient rule schedules.

3. **Guards and Atomicity**: rules executing it isolation with explicit guards is a natural way to think about parallelism, since the semantics of composition are very clear.

4. **Functional Language Features**: purely functional code (no side effects) lends itself naturally to describing parallel computation, easing the burden of optimization.

The code used to describe the logic in the rules is specified in the functional paradigm, familiar to many programmers (though all actions or side-effects take place in parallel). Conventional compiler techniques can be leveraged to optimize these code blocks once a rule schedule has been established. What the BTRS compiler adds to this picture is the exploitation of the high-level information inferred from the structure of the rules and user-supplied scheduling constraints.

MIT has a rich tradition of parallel programming language research. Of great relevance are the [relatively recent] languages pH[1] (an explicitly parallel functional language) and its close relative Id[12]. Id is an explicit data-flow language implying single assignment. For the sake of generating efficient code at the computational nodes, M-Structures and I-Structures were used to get around this, much in the same way that Haskell uses Monads to allow for the use of the imperative programming paradigm. In the case of BTRS, we can embed arbitrary computation within rules, and are not subject to this kind of restriction. StreamIT[14] develops a similar network of computation nodes (Filters, as they are referred to), though it is a language which has been specialized towards streams. In BTRS, a Filter could be represented as either a single rule or a module, though unlike StreamIT, BTRS does not require the specification static rates of consumption and production. These approaches have achieved varying degrees of success in tackling the challenges of expressing parallelism. The semantics of these languages differ quite significantly, though insight into their implementation is quite valuable.

Perhaps the closest semantic relative to BTRS is the language Unity [2], which is based on Dijkstra's notion of non-deterministic guarded commands [6]. The table in Figure 7-1 enumerates the four defining features of BTRS and lists other explicitly parallel languages which share those features.

Note that languages with explicit communication models are not necessarily non-deterministic. CSP employs sequential processes exchanging messages, and Occam is an implementation based directly on Hoare's CSP. Ada is not necessarily thought of as a parallel language, but it does define tasking and rendezvous features as language primitives in parallel contexts.

Non-determinism	Explicit Communication	Functional	Atomicity
Unity	CSP	MultiLisp	Transactional Memory
PCN	Occam	QLisp	Unity
CSP	Ada	Id	
PARLOG	Orca	Sisal	
	Concurrent Smalltalk		
	Cantor		
	StreamIT		

Figure 7-1: Features of Explicitly Parallel Languages

Orca defines shared objects which use broadcasting and Cantor was designed for use in programming fine-grained hardware supported parallelism. Lastly, Concurrent Smalltalk has a notion of distributed data structures with methods which are used for communication.

The Liquid Metal project has a surprising amount in common with BTRS. Semantically it is very different since it is Java based and sequential. Like BTRS, though, the designers are trying to create a system where modules can be arbitrarily synthesized to either hardware or software [9]. The interesting feature is that particular restrictions on HW-synthesis candidates give a natural translation of these modules to StreamIT. Liquid metal also suffers from a more restrictive computational model for those modules which could potentially end up on the FPGA.

Bibliography

[1] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A parellel dialect of Haskell. In *Haskell Workshop*, 1995.

[2] K. Mani Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

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

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

[5] Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, and Muralidaran Vijayaraghavan. Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA. In *MEMOCODE*, pages 97–100, 2007.

[6] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[7] Kermin Fleming, Myron King, Man Cheuk Ng, Asif Khan, and Muralidaran Vijayaraghavan. High-throughput Pipelined Mergesort. In *MEMOCODE*, pages 155–158, 2008.

[8] ITU-T Video Coding Experts Group. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, May, 2003.

[9] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric M. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP*, pages 76–103, 2008.

[10] Daan Leijen. Parsec, a fast combinator parser. 2001.

[11] Chun-Chieh Lin. Implementation of H.264 in Bluespec System Verilog, 2007.

[12] R. S. Nikhil. Id language reference manual (version 90.1). Technical Report 284-2, 1991. citeseer.ist.psu.edu/nikhil91id.html

[13] Iain E.G. Richardson. In *H.264 and MPEG-4 Video Compression*. John Willey & Sons, 2003.

[14] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, pages 179–196, 2002.