Modular Scheduling of Atomic Actions

Daniel L. Rosenband, Jacob B. Schwartz, and Arvind
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139, USA
{danlief, quark, arvind}@lcs.mit.edu

Abstract

Sets of atomic actions in the form of a Term Rewriting System (TRS) have been used to describe complex hardware systems. Besides being convenient for "behavioral modeling," such descriptions are suitable for both formal verification and high-quality hardware synthesis. Recent work has provided an object-oriented front-end for packaging these atomic actions into modules where each module has its own state, rules and interface. This paper presents an algorithm to perform modular compilation given the scheduling constraints on the interface methods of primitive modules. In addition, the designer has the option to overrule the scheduling constraints that the compiler derives because the designer may have knowledge that the compiler cannot have. Such a modular synthesis flow not only reduces compile-times dramatically but also results in designs that are closer to the designer's intent. A pipelined processor is used to illustrate the benefits of this flow on the pipeline schedule and the compile time in comparison to the compilation that flattens all the modules first.

1. Introduction

Arvind and Shen showed how to use atomic actions in the form of Term Rewriting Systems (TRS) for behavioral modeling of complex processors and cache coherence protocols [1]. They also showed how formal methods could be applied to prove properties of such systems [14, 13]. Hoe and Arvind further showed that good quality structural RTL could be generated from such descriptions [11, 10]. The innovative part of this work was showing how to schedule many of the atomic actions in parallel while maintaining an execution model that makes it appear as though all actions executed in some sequential order.

An advancement of enormous practical importance was made by Lennart Augustsson of Sandburst Corporation when he embodied this model of describing hardware into a high-level, object-oriented language called Bluespec [6]. An object in Bluespec represents a hardware module with internal state, rules to manipulate the state, and an interface (a set of methods) through which other modules can observe and manipulate this state. This language has been used successfully by Sandburst for behavioral modeling of a complex chip-set for high-speed routers.

The current Bluespec compiler flattens the modules in a design until all method calls are to Verilog or C primitives. Only then does the compiler begin applying the scheduling algorithms. The drawback of this approach is that schedules are often too restrictive and the compile times grow at least quadratically in the number of rules. The main contribution of this paper is a modular compilation algorithm, which solves both of these problems. We show how to generate a matrix of scheduling information for the interface methods of a module given the same information regarding the module interfaces whose methods this module calls. If the compiler is not able to derive the correct (i.e., most liberal) scheduling constraints for a set of interface methods, we show how the user is able to override the compiler through user annotations that carry proof obligations. One can think of scheduling information associated with an interface as an extension of the usual type signature of a module. Similar to a type signature, the scheduling information can be either derived automatically or be prescribed. We think that for good modular hardware design both these properties are essential.

The biggest impact of modular compilation is that a designer can build highly-parameterized modules whose interface scheduling or concurrency properties are not left to the vagaries of the compiler. For example, we can build a FIFO which permits concurrent enqueue and dequeue operations on it. Such a FIFO can then be used in larger designs without having to worry about concurrency issues relating to its interface methods. The current Bluespec compilation scheme does not provide any such assurance except for primitive modules. This often causes designers to spend time figuring out why rules are not firing concur-

rently. Thus, modular compilation provides a powerful tool for designers to express their designs in a way that meets their intent. At the same time modular compilation drastically reduces compile times by not requiring each rule in every module to be compared with every rule in every other module when determining scheduling conflicts.

1.1. Related work

There has been a strong interest in high-level design languages which can bridge the gap between behavioral modeling and efficient hardware synthesis. Synthesis of specifications in the synchronous, concurrent language Esterel is being pursued in several directions, well documented in [5]. Many other efforts involve extending the C syntax with additional mechanisms for expressing hardware-related information, such as SystemC [9] and SpecC [8]. This stems from a desire to embed support for C and C++ models into existing HDL design flows, as with the Scenic design environment [12].

The benefit of an operation-centric rule-based description of hardware is that it can also be helpful in formal verification. Much research is involved in verifying that a circuit is a valid implementation of an operational specification [3], particularly as designs become more complex, with more pipelining and out-of-order processing. For example, Velev [15] showed that rewrite rules can be used to verify modern processor architectures. Formal verification has also been applied to synthesizable designs in SystemC [7, 4], but we feel that the clear execution model of atomic rules can lead to better analysis while still producing efficient hardware.

1.2. Paper organization

In Section 2, we introduce hardware descriptions using atomic actions through a simple two stage processor example. In Section 3, we present a highly parameterized FIFO which we want to implement using the high-level features of Bluespec and then instantiate in our two stage processor. In Section 4, we informally analyze what level of atomic action scheduling concurrency we achieve in the traditional flat compilation flow. We also hint at the way modular compilation can avoid the problems we encounter in the flat flow. Section 5 provides a precise definition of our modular compilation algorithm. Section 6 illustrates how we generate circuits after having derived the atomic action execution schedule using the algorithm from Section 5. Section 7 presents results and we conclude in Section 8.

2. Two-stage processor description using atomic actions

We describe a two-stage processor pipeline to familiarize the reader with a hardware description that uses atomic actions. Our processor implements four instructions (i.e., Add, Branch on Zero (Bz), Load, and Store) all of which operate on register operands. Figure 1 shows the type signature of the instruction and instruction template formats. (The instruction template is the format of the instruction in the pipeline stage.)

```
data RName =    R0  | R1  | ... | R31

type Src, Dest, Cond, Addr, Val = RName

type Value = Bit 32

data Instr = Add    Dest Src Src
           | Bz     Cond Addr
           | Load   Dest Addr
           | Store  Val  Addr

data InstrTemplate = EAdd   Dest  Value Value
                   | EBz    Value Value
                   | ELoad  Dest  Value
                   | EStore Value Value
```

Figure 1. Instruction formats

As shown in Figure 2, we divide the processor into a "Fetch & Decode" (FD) stage and a "Execute" (EX) stage and connect them with a FIFO. (We generally implement pipeline buffers as FIFOs to maintain an asynchronous interface between pipeline stages. A FIFO of size one will result in the same pipeline as if a register had been used instead. Using a FIFO as a pipeline stage also makes for an interesting example when we discuss the modular compilation flow.) The FD stage reads the instruction from memory, decodes it, reads the operands from the register file and enqueues the resulting instruction template into the pipeline FIFO bd. Each time an instruction is fetched, the program counter (pc) is speculatively updated to pc + 1 so that the next instruction can be fetched in the following cycle. The "Execute" (EX) stage performs the desired operations on the instruction templates in bd. In the case of Add, Load, and Store, this is straightforward: the desired computation is performed and if necessary, the result is written to the register file or data memory. The template is also dequeued from the bd FIFO. In the case of Branch on Zero (Bz), if the branch operand is non-zero, the instruction is simply dequeued from bd. However, if the operand is zero, then the pc is set to the branch target address and the instructions that we had speculatively fetched are thrown away by clearing the bd FIFO.

A small complication in the FD stage is that before the operands are fetched from the register file, one needs to make sure that none of the instruction templates in the bd FIFO intend to modify the registers being fetched. If such a condition exists then the pipeline needs to be stalled. Such

a check requires searching the contents of the FIFO. The nofind1 and nofind2 methods provide an interface to such a search function.

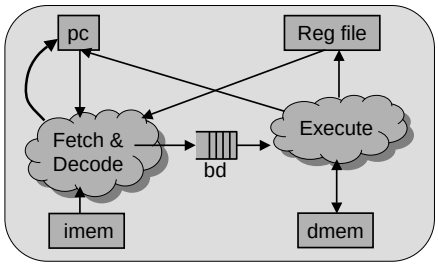

Figure 2. Two-stage pipeline

2.1. Two-stage processor code

Figure 3 contains stylized code that implements the two-stage processor. The first part of the module definition contains the state declarations and instantiations of appropriate modules to create this state. In this example, the state consists of the data and instruction memories (instMem and dataMem), the register file (rf), the program counter (pc), and the pipeline stage FIFO (bd). Following the state declarations is the rules section, where each rule specifies how and when state should be updated atomically. All rules take the form:

```
<rule_name>:when <firing condition> ==>
              <calls to module methods to modify state>
```

We will assume that the register file implementation permits two register reads and one write during an atomic action (cycle). For now, we will also assume that the FIFO implementation allows simultaneous enqueuing and dequeuing and up to two nofind searches every cycle. One thing to note is that the rules do not explicitly check for a FIFO full or FIFO empty condition; these conditions are implied by the FIFO definition and propagated implicitly by the compiler.

2.2. Analysis of concurrent rule firings

The execution model for a set of rules can be simply stated as follows:

> Given a set of rules and an initial state s
> While (some rules are applicable to s)
> - choose an applicable rule
> - apply the rule atomically to s

```
mkTwoStage :: Module (Empty)
mkTwoStage =
  module
   instMem  :: MemoryIF <- mkMemory 1024 Instr
   dataMem  :: MemoryIF <- mkMemory 1024 Value
   rf       :: Array RName Value <- mkArray
   pc       :: Reg (Bit 10) <- mkReg 0
   bd       :: S_FIFO_2Stage <- mkS_FIFO_2Stage

    /* **********************  Fetch & Decode Rules ********************** */
   let
    inst     = instMem.get pc
   rules
    fd_add:when (inst == Add rd ra rb) & (bd.nofind1 ra) & (bd.nofind2 rb) ==>
               bd.enq (EAdd rd (rf.read ra) (rf.read rb))
               pc := pc + 1

    fd_bz:when (inst == Bz cd addr) & (bd.nofind1 cd) & (bd.nofind2 addr) ==>
               bd.enq (EBz (rf.read cd) (rf.read addr))
               pc := pc + 1

    fd_ld:when (inst == Load rd addr) & (bd.nofind1 addr)  ==>
               bd.enq (ELoad rd (rf.read addr))
               pc := pc + 1

    fd_st:when (inst == Store v a) & (bd.nofind1 v) & (bd.nofind2 a) ==>
               bd.enq (EStore (rf.read v)  (rf.read addr))
               pc := pc + 1

    /* **********************  Execute Rules ********************** */
   let
    instTemplate     = bd.first
   rules
    ex_add:when (instTemplate == EAdd rd va vb) ==>
               rf.write rd (va + vb)
               bd.deq

    ex_bz_taken:when (instTemplate == EBz cv av) & (cv == 0) ==>
               pc := av
               bd.clear

    ex_bz_not_taken:when (instTemplate == EBz cv av) & (not (cv == 0)) ==>
               bd.deq

    ex_ld:when (instTemplate == ELoad rd av) ==>
               rf.write rd (dataMem.get av)
               bd.deq

    ex_st:when (instTemplate == EStore vv av) ==>
               dataMem.put av vv
               bd.deq
```

Figure 3. Two-Stage Pipeline State & Rules

Thus, even if a nondeterministic choice is made in selecting the rule to be applied, the execution of the system will appear as though rules were applied to the system in some sequential order. Each rule execution is atomic with respect to the other rule executions. The reader should be able to convince himself that in our processor example, we obtain a valid execution no matter what order the rules are applied in.

However, a pipeline design can hardly be said to meet its specifications unless both stages in the pipeline can fire in parallel. We make the following assertions informally:

- The four rules in the FD stage are mutually exclusive.
- The five rules in the EX stage are mutually exclusive.
- If simultaneously enabled, any rule from the FD stage can fire concurrently with any rule in the EX stage except the ex_bz_taken rule because FD and EX stage rules do not affect the same state elements. Even during simultaneous execution, the sequential and atomic execution model will be preserved.

This is the best we can hope to do as far as concurrent firings go. However, our informal reasoning relies crucially

on the assumption that enqueue and dequeue operations can be performed concurrently on a FIFO that is neither empty nor full. We will revisit this assumption after giving an implementation of the FIFO in the next section.

3. The FIFO module

One of the advantages of Bluespec is that it allows highly parameterized modules to be created. These modules can be used repeatedly within a design, and in many cases across different designs. Figure 4 shows the code of a parameterized searchable FIFO. The number of entries in the FIFO is set by the parameter sz. The width in bits of FIFO entries is determined implicitly by t, the type of the elements to be stored in the FIFO. Both of these parameters are specified when the FIFO is instantiated. The FIFO interface contains enqueue (enq), dequeue (deq), first, clear and find methods. The find method takes a function f as a parameter and applies f to each FIFO element. It returns True if f(e) is True for some element e in the FIFO. Otherwise it returns False.

```
mkS_FIFO :: Integer -> Module (S_FIFO t)
mkS_FIFO sz =
  module
    rs :: List (Reg t)
    rs <- map mkReg (upto 0 sz)

    head :: Reg (Bit (Log sz)) <- mkReg 0
    tail :: Reg (Bit (Log sz)) <- mkReg 0

    let
      maxptr = sz - 1
      minptr = 0

      incr ptr = if (ptr == maxptr) then minptr else ptr + 1
      decr ptr = if (ptr == minptr) then maxptr else ptr - 1

      notEmpty = not (head == tail)
      notFull  = not ((incr tail) == head)

      get i   = (select rs i).read      -- select yields a register
      put i v = (select rs i).write v

      findfunc :: (t -> Bool) -> (Bit (Log sz), Bit (Log sz)) -> Bool
      findfunc f (hd, ptr) =
        if (ptr == hd) then False
        else let
               new_ptr = decr ptr
               elem = get new_ptr
             in
               if (f elem) then True
               else findfunc f (hd, new_ptr)

    interface
      enq x = put tail x
              tail := incr tail
      when notFull

      first = get head
      when notEmpty

      deq = head := incr head
      when notEmpty

      clear = head := 0
              tail := 0

      find f = findfunc f (head, tail)
```

Figure 4. Parameterized S_FIFO

The implementation in Figure 4 uses sz + 1 registers for a FIFO that can hold sz elements. It also uses two additional registers head and tail: head points to the oldest element in the FIFO while tail points to the slot where the next element is to be enqueued. Even though the FIFO is represented as a list of registers in the program, all the operations associated with the list are eliminated at the first stage of compilation.

Two aspects stand out when comparing this module to what a comparable Verilog module might look like. First, the module is easily parameterized. The second property that stands out is the sophistication of the find method. It is a recursive function that takes a function as input. Through partial evaluation during compile time, the compiler is able to generate efficient hardware for this type of method.

```
mkS_FIFO_2Stage :: Module (S_FIFO_2Stage InstTemplate)
mkS_FIFO_2Stage =
  module
    f :: S_FIFO InstTemplate
    f <- mkS_FIFO 2

    let
      findf r it  =
        case it of
          EAdd rd va vb -> (r == rd)
          EJz  vc va    -> False
          ELoad rd va   -> (r == rd)
          EStore vv va  -> False

    interface
      enq v     = f.enq v
      first     = f.first
      deq       = f.deq
      clear     = f.clear
      nofind1 r = not (f.find (findf r))
      nofind2 r = not (f.find (findf r))
```

Figure 5. S_FIFO Instantiation

3.1. FIFO instantiation

To instantiate a FIFO we need to specify the number and the type of its entries and the function to be passed to the find method. This is done by providing a wrapper module as shown in Figure 5. The wrapper module's interface has the usual enq, first, deq, and clear methods plus two nofind methods. The FIFO is instantiated to hold two entries of type InstTemplate. The find function (findf) checks if the instruction template contains a "destination" register, and if so, checks if it is the same as the register that was passed to one of the nofind methods. We need two nofind methods since some instructions, for example Add, contain two operands. A check needs to be performed for each operand to see if it is being written to by an instruction in the bd FIFO.

The FIFO could have been instantiated directly in the two stage processor code. Since we wanted to explore modular compilation, it made sense to instantiate the FIFO in this new wrapper. Since all the parameters, functions, and number of ports for each method are specified in this wrapper, we are able to compile it and then use it as a precompiled module in the two stage processor.

Given the expressiveness of the language, it is clear that designers will create libraries of modules that are used many times. Just through our work on processor design exploration, we have created a library of FIFOs that range from a standard FIFO to the type of FIFO we would need to implement bypasses in a processor. Clearly, other modules

such as register files, completion and reorder buffers, etc. have also been explored. We view these modules as a key advantage of the language. That is what has motivated us to examine how these modules affect the scheduling in the system as a whole.

4. Rule scheduling and conflict matrices

One of the main challenges of synthesizing efficient hardware from atomic actions is to find a maximal set of rules that can execute in every cycle. Which rules execute in a cycle is determined by a scheduler circuit that takes as input the set of rules whose firing condition is satisfied in that cycle and outputs the set of rules that can actually execute in the same cycle. This section outlines how the scheduling relationship between rules can be determined with the help of conflict matrices. Using the FIFO and two stage processor example we also illustrate why these scheduling relations are critical to the performance of the system and how modular compilation can help us achieve our scheduling goals. The key property to keep in mind during this discussion is that we are looking for a relaxed schedule, but we need to maintain the appearance of sequential and atomic execution of the rules.

4.1. Conflict matrices

The compilation flow that Hoe and Arvind as well as the Bluespec compiler use assigns a conflict matrix to primitive Verilog modules. These conflict matrices allow the compiler to derive the type of concurrency that we informally described in Section 2.

Table 1. Conflict Matrix for Register

	read	write
read	ne	$<$
write	$>$	$<$ and $>$

Table 1 shows the conflict matrix for the read and write methods of a register. This matrix tells us that reading the value of the same register in two rules has no effect (ne) on the scheduling relationship between those rules.

Similarly, the matrix tells us that if rule r1 reads the value of a register and r2 writes a value to the same register in the same cycle, then it must appear as though r1 occurs before r2 (r1 $<$ r2). If other method invocations within these rules indicate r2 $>$ r1, then r1 and r2 cannot execute simultaneously; they are conflicting.

In the case where both rules r1 and r2 write to the register, the rules can execute simultaneously as long as there are no other conflicting method invocations. However, we need to choose an order that sequential execution corresponds to. If we compose them such that r2 $>$ r1, then it must appear as though r2 happens after r1. In this case r2 determines which value gets written to the register since it occurs after r1.

4.2. FIFO method conflicts in the processor

Table 2 provides the conflict matrix that we desire for the FIFO in the two stage processor. Unfortunately, when we do some informal analysis of the FIFO implementation given in Figure 4, it becomes clear that the compiler cannot derive such a conflict matrix. Instead, it derives the conflict matrix in Table 3. For the most part these tables are identical. However, a key entry, the enq and deq relationship, is different in the two tables. We want the relationship between enq and deq to be "no effect" (ne), but the compiler derives a conflicting (c) relationship.

Table 2. Desired Conflict Matrix for FIFO

	enq	first	deq	clear	nofind1	nofind2
enq	c	ne	ne	$<$	$>$	$>$
first	ne	ne	$<$	$<$	ne	ne
deq	ne	$>$	c	$<$	$>$	$>$
clear	$>$	$>$	$>$	ne	$>$	$>$
nofind1	$<$	ne	$<$	$<$	ne	ne
nofind2	$<$	ne	$<$	$<$	ne	ne

By inspection we can see why the compiler derives a conflicting relationship. The enq method modifies the tail pointer and reads both head and tail pointers in its implicit (when) condition. deq modifies the head pointer and also reads both head and tail pointers in its implicit condition. The fact that enq writes to tail and deq reads the tail creates a deq $<$ enq constraint. Similarly, deq writing to head and enq reading that value creates an enq $<$ deq constraint. These constraints are contradictory, and hence the compiler deduces that these two methods conflict.

Table 3. Derived Conflict Matrix for FIFO

	enq	first	deq	clear	nofind1	nofind2
enq	c	$>$	c	$<$	$>$	$>$
first	$<$	ne	$<$	$<$	ne	ne
deq	c	$>$	c	$<$	$>$	$>$
clear	$>$	$>$	$>$	ne	$>$	$>$
nofind1	$<$	ne	$<$	$<$	ne	ne
nofind2	$<$	ne	$<$	$<$	ne	ne

The consequence of enq and deq conflicting is that a rule that enqueues into the FIFO can never execute simultaneously with a rule that dequeues from the FIFO. In the processor example this has a disastrous impact on performance since it means that a FD rule can never execute simultaneously with a EX rule. The processor would fetch an instruction in one cycle, execute it in the next cycle, then fetch another instruction in the third cycle, etc. This type of behavior defeats the whole idea of a pipelined design.

However, as programmers of the FIFO module, we know that enq and deq should not conflict. This is because we know that an enq operation will never cause the FIFO to become empty and in turn, a deq to become invalid. A similar argument can be used to show that a deq can never cause an enq to become invalid.

Our strategy for fixing the above problem is to let the designer specify the conflict matrix entries for a module. When these entries are less constrained than what the compiler may be able to deduce, then the annotations carry proof obligations on the part of the designer. We would like the compiler to verify each conflict matrix entry, but it should accept them as true unless it can prove otherwise. Thus, to make our two stage processor pipeline work correctly, we use Table 2 entries as prescriptive.

One of the reasons we introduced a modular compilation approach is because these annotations are hard to propagate in the current compilation flow. The current compilation flow flattens the modules and performs as much partial evaluation as possible before computing the conflict information. After flattening, the FIFO method calls turn into register reads and writes, and hence any correspondence between the method annotations and the register operations is hard to derive.

In the modular approach, we compile the FIFO (really the FIFO_2Stage) on its own. This then becomes a primitive module and the interface method conflict annotations are propagated to the module that is instantiating the FIFO — the two stage processor in this case. Thus, the fact that enq and deq are conflict free becomes easy to propagate to the rules that call the enq and deq methods. This annotation also has no effect on the circuit that is generated for the FIFO itself since nothing in the FIFO circuit prevents enq and deq from happening simultaneously. Thus, through modular compilation we are able to express the FIFO in a high level language and still obtain the expected scheduling — a real pipeline. As we show later, it also turns out that this compilation strategy is significantly faster than the flat flow. The next section provides the detailed and more precise algorithm on how we perform the modular compilation.

5. Algorithm

In this section we present an algorithm for compiling a hierarchy of modules. The algorithm assumes a hierarchy which can be represented as a directed acyclic graph (DAG). Nodes in the graph represent modules. Edges point from modules which make method calls to the modules whose methods they invoke. There is one top-level module whose node has only out-going edges. Primitive modules, such as registers, do not make any method calls and thus have no out-going edges in the graph.

The previous compilation strategy has been to flatten the hierarchy before scheduling is performed. This means that all rules, state, and interface definitions of submodules are inlined into the parent modules until a single collection of state and rules is accumulated. Then, scheduling of the rules in this single module is performed. The schedule is constructed based on the constraints of the primitive modules. These primitive module constraints are in the form of conflict matrices as was described in section 4.

In our modular compilation flow we schedule and generate code for each module in the hierarchy, independent of any parent modules. We start from the leaves and work up, propagating scheduling information about a module's methods to the modules which call those methods. We also allow the user to prescribe the scheduling information at any of the module boundaries.

5.1. Deriving method relationships

The core of our modular compilation flow takes a module that has all the scheduling information for the methods of the modules that it invokes and turns the module into something that looks like a primitive module to the rest of the design. Essentially, it takes the conflict matrices of all modules that it calls methods from. Using these constraint matrices it generates a circuit that represents the state changes and method calls, and also passes a new constraint matrix to the modules that invoke methods within it. We will touch on circuit generation in section 6, but it is not much different from the current flat circuit generation methods. What differs in our flow is that we construct the conflict matrix at each module boundary and propagate that through the hierarchy. The reader should recall that these conflict matrices are used to generate a scheduler that decides which rules can execute simultaneously in a design.

The conflict matrix entries that we support are conflicting ("c"), sequentially composable in one direction ("$<$" or "$>$"), sequentially composable in both directions but with different outcomes depending on direction chosen ("$<$ and $>$"), and conflict free ("ne" for "no effect"). Rules which are conflict free are composable in both directions but the order that they are applied in does not effect the outcome.

```
DERIVERELATIONSHIP (calls_1, calls_2) =
  do result ← "conflict free"
     for each method s_i . m_a ∈ calls_1
       for each method s_j . m_b ∈ calls_2
         if s_i == s_j
         then do rel ← PCM_{s_i} (m_a, m_b)
                  result ← lookup (result, rel) in Table 4
     return result
```

Figure 6. Algorithm for deriving basic conflict relationships

Table 4. Formula for incrementally restricting a rule relationship

result so far	relationship of next pair	new result
c	*any*	c
<	c	c
	>	c
	otherwise	<
>	c	c
	>	>
	otherwise	>
< and >	c	c
	>	>
	<	<
	otherwise	< and >
ne	*x*	*x*

Figure 6 shows the algorithm to derive the scheduling relationship between two rules or interfaces. It takes as input the set of all methods invoked by the first rule (or interface) and the set of all methods invoked by the second rule (or interface). The procedure also has access to PCM_s, the prescriptive conflict matrix, for all submodules s. PCM_s specifies the conflict relationship of all interface methods of submodule s, whether user specified or compiler derived. The procedure DERIVERELATIONSHIP returns the relationship between the two rules (or interfaces), based on pairwise comparison of their method calls. The procedure starts by assuming that two rules (or interfaces) are conflict free and then restricts the possibilities for concurrency as each pair of methods between the rules is considered. If the procedure eliminates all possibilities of concurrency, then it reports that the rules are conflicting. If, after all pairs of methods have been considered, there still remain possibilities for concurrent execution, the procedure reports that relationship.

5.2. S_FIFO enq and deq method relationship

Table 5 shows the statements from the definition of the enqueue and dequeue methods for the S_FIFO. On the left are the statements in enqueue and the register methods that are invoked by those statements. On the right are the statements and methods for dequeue.

Table 5. S_FIFO enq and deq method calls

enq statement	enq method call	deq statement	deq method call
put tail x	tail.read	head := incr head	head.read
	rs[tail].write		head.write
tail := incr tail	tail.read	when notEmpty	head.read
	tail.write		tail.read
when notFull	head.read		
	tail.read		

To derive the relationship between the enq and deq methods using the algorithm just described, we begin by assuming the methods are conflict free. We then compare all pairs of methods from column two with methods from column four. We begin by examining the tail read from enqueue with the head read from dequeue. Since they are not acting on the same state, they do not restrict the relationship. We move on to the next method call in dequeue until we reach another method on the same state, this time the tail read in the `when`-clause. Two register reads are conflict free, so again we have not restricted the relationship. It is not until we reach tail write in enq and tail read in deq that we discover that the methods cannot be scheduled such that deq happens second. This is because the tail read method is only composable before the tail write. Thus, we have the constraint deq $<$ enq. We continue to compare other pairs of methods until we reach head read in the enq method and head write in the deq method. This tells us that deq $>$ enq. Looking up in table 4 what the result of $<$ with a new relations $>$ is, we see that the methods must be conflicting.

5.3. Hierarchical Compilation

To compile an entire design, we first build a graph which represents the call hierarchy of the modules in the system. Then we traverse the hierarchy, starting with the modules at the bottom which make no calls to other modules besides the primitives. We are provided the conflict matrices for the primitive modules, so we can perform the DERIVERELATIONSHIP procedure on all pairs of interface methods for the bottom modules to compute their conflict matrices. We can also compute the schedules for these modules and generate their circuits. Once we have the conflict information for the bottom modules, we can move to the next higher level and compute the matrices for those modules. This is repeated until a circuit is generated for the top-level module.

This procedure is given formally as COMPILE in Figure 7. The procedure takes the top-level module as input and recurses down through the hierarchy until the leaves are reached. Then, for each submodule, the procedure computes *DCM*, the derived conflict matrix. This matrix is compared against the prescribed conflict matrix, *PCM*, given by the user. The user annotations take preference over the de-

rived conflict relationships when we combine them. The reconciled *PCM* is then used to schedule the local rules and generate the circuit. Finally, the *PCM* is passed to the parent module where it is used to derive the *DCM* for the next level.

```
COMPILE (s) =
  do for each module s_i invoked by s
       COMPILE (s_i)
     for each r_a, r_b ∈ rules and interface methods of s
       CM(r_a, r_b) ← DERIVERELATIONSHIP (calls(r_a), calls(r_b))
     for each m_a, m_b ∈ interface methods of s
       DCM_s(m_a, m_b) ← CM(m_a, m_b)
     PCM_s ← RECONCILE (PCM_s, DCM_s)
     GENERATESCHEDULER (s, PCM, CM)
     GENERATECIRCUIT (s, PCM, CM)
```

Figure 7. Algorithm for hierarchical compilation

Because the procedure for deriving relationships compares two state elements for equality, we must be sure that sufficient inlining and partial evaluation has been performed so that there is no aliasing. Also note that when this procedure is applied to a rule, the set of methods that we extract are those methods invoked in the explicit firing condition and in the action of the rule. The implicit conditions that are carried with method calls do not impact the calculation of conflict relationships since they are single bit independent boolean values when we perform modular compilation.

5.4. Additional Optimizations

We have presented a basic system which allows the compiler to perform concurrent scheduling of rules which refer to separately compiled modules. There are infrequent cases where additional improvements can be made to achieve more concurrency and simpler schedulers.

A case where the current conflict matrix entries are not descriptive enough is a register file that has a single read port, a single write port and is created using primitive registers. A conflict can arise when read and write are invoked with the same register address. However, when read and write addresses are different, the two methods should be conflict free. The current conflict matrix has no way to describe a scheduling relationship that is dependent on method parameters. Extending our compilation strategy to accommodate user annotations that allow for parameterized conflict relationships is not overly difficult. However, we believe this is a rare case and the details beyond the scope of this paper.

Besides changing the conflict information, we can also add information about implicit condition signals to the conflict matrices. For example, we could indicate when the implicit condition signals for two methods are mutually exclusive. This information would help the compiler deduce the mutual exclusion of two rules containing calls to those methods, thereby simplifying the scheduler. Again, we believe this to be a highly specialized case.

6. Circuit Generation

Hoe [10] showed that a system of atomic actions can be implemented efficiently as a finite state-machine. To compile a module to hardware, we generate the state elements and the logic which updates the state on each cycle. Rules consist of a firing condition, which historically has been called π, and a set of actions, δ, that are applied to state elements or modules. A scheduler circuit is used to generate a ϕ signal for each π. The ϕ signal determines if a rule's delta function is applied. If several rules that are conflicting are enabled (their π is True), then the scheduler needs to pick one of them so that only one of their ϕ signals is asserted.

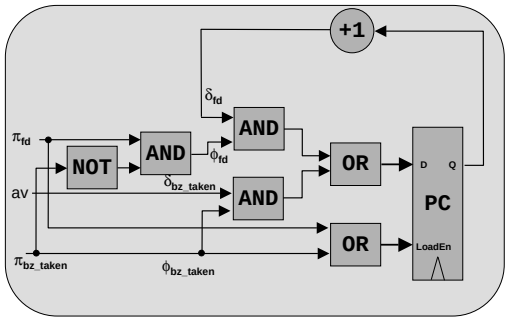

Figure 8. Program Counter Update Logic

The enable signal for a interface method call is the OR of the ϕ signals for all rules whose actions invoke that method. If any one of those rules is selected to fire, then the method is going to be invoked. The signals for the method's arguments require additional logic because the rules might call the method with different arguments. Arbitration logic, which depends on the rule conflict relationship, is necessary to select which rule's value is used. If the rules are sequentially composable, then the last rule to fire is the only rule whose effect is visible. Hence, that last rule would be the rule that passes the value to the method invocation. If the rules are conflict free, then the method is idempotent and the arguments from any of the fired rules can be chosen (though usually idempotent methods do not take arguments). If the rules are conflicting, the arbitration logic will assume that the scheduler will only ever pick one rule to fire and the arbitration logic can select the arguments from any fired rule that it finds.

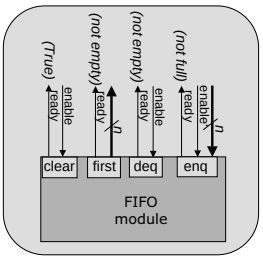

Figure 9. FIFO hardware interface

We present an example of the use of these signals in figure 8. The figure shows the program counter update logic that we get when generating hardware for the two-stage processor. The two possible delta functions that can be applied to the pc are "pc + 1" and "av" (a field from the Bz instruction in the execute stage). Which delta function, if any is applied to the pc is determined by the scheduler. As we expect, the "pc + 1" value is used unless the bz_taken_rule is enabled, in which case the "av" value is used.

Figure 9 shows what a FIFO module interface looks like after hardware has been generated for it. Each interface method has a ready output signal that corresponds to the method's implicit condition. For interface methods that update state within the FIFO (for example clear), it accepts an enable signal which indicates that the method should be executed. The interface method enable signal is the equivalent to the π firing condition in a rule. Clearly, method interfaces can also accept parameters (for example enq) or return data (for example first). After modular compilation, all modules have these types of hardware interfaces.

7. Results

We obtain two very positive results by using a modular compilation flow. First, the flow allows us to add scheduling annotations to module boundaries. These annotations, when carried through to the instantiating modules allow us to obtain the scheduling results that we expected when creating the modules. The second improvement is that compile times are dramatically lower than in a flat flow.

Table 6 summarizes scheduling and compile time results from experimentation that we performed on several processor models. The processors that we examined were the two stage processor presented in this paper, a five stage processor with bypasses that implements the same ISA as the two stage processor, and a five stage fully bypassed MIPS processor core. The MIPS processor core implements most instructions of the MIPS-II instruction set (TBD add reference). We have verified the correctness of our MIPS core by running binaries on it in simulation. We implement the FIFO in BlueSpec in all cases, not as Verilog primitives.

The first column of table 6 lists the processor being compiled. We compiled all processors using the conventional flat compilation flow. We also compiled all processors using a modular flow that compiles the FIFOs (with search / bypass function) separately. In the case of the MIPS core we also ran an experiment in which the register file along with the FIFOs are each compiled as separate modules.

The second column of the results table indicates whether the processor pipeline acts as a real pipeline or whether only one stage can execute at a time. As we expect, flat compilation derives a bad schedule wheras the modular flow creates the correct pipeline.

Table 6. Flat vs. Modular Compilation

Processor	Real Pipeline	Partial Eval.	Scheduler	Total
2-Stage Flat	No	0.7s	1.0s	3.2s
2-Stage Modular	Yes	0.1s	0.1s	2.0s
5-Stage Bypass Flat	No	26.8s	Opt. OFF	29.4s
5-Stage Bypass Modular	Yes	0.9s	0.2s	3.6s
MIPS Flat	No	1035.8	Opt. OFF	1052.2s
MIPS Modular FIFO	Yes	46.0s	218.1s	275.8s
MIPS Modular FIFO + RF	Yes	21.9s	1.8s	35.7s

The two largest compilation phases are partial evaluation and scheduling. The partial evaluation phase expands the code by inlining all functions and modules, performs partial evaluation wherever possible, unrolls recursive calls, etc. The scheduling phase of the compiler generates the scheduler – decides which rules are mutually exclusive, conflicting, etc. In both of these phases the modular flow is significantly faster than the flat flow. This is largely due to fewer rules needing to be compiled when using the modular flow and expression sizes getting reduced. In the scheduling phase not all optimizations could be turned on in the flat flow as expression sizes got too large for some algorithms that are implemented using BDDs [2]. As expected, the total compile time is also dramatically less in the modular flow.

8. Conclusion

In this paper we presented an algorithm for modular compilation of atomic actions. This compilation strategy greatly improves compile times, which in turn makes experimentation with larger designs more practical. Through the use of scheduling annotations at module boundaries we were also able to build libraries using the full power of the BlueSpec language, without relaxing requirements on scheduling efficiency. Previously, these blocks had to be implemented in Verilog in order to allow the proper schedule to be derived. The combination of compile time improvements and ability to build libraries native to the BlueSpec language fundamentally strengthens the usability of

the infrastructure. We can now take full advantage of the language, build intellectual property libraries using it, and still get the performance we expect.

We would like to thank Lennart Augustsson, Mieszko Lis, and Rishiyur S. Nikhil for providing us with the BlueSpec compilation infrastructure.

References

[1] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on Modelling and Validation of Microprocessors*, 19(3):36–46, May/June 1999.

[2] Randal Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1994.

[4] Rolf Dreshsler and Daniel Große. Reachability analysis for formal verification of SystemC. In *Euromicro Symposium on Digital System Design (DSD'02)*, September 2002.

[5] Stephen A. Edwards. High-level synthesis from the synchronous language Esterel. In *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, New Orleans, Louisiana, June 2002.

[6] Lennart Augustsson et al. Bluespec: Language definition, 2001. `http://bluespec.org`.

[7] F. Ferrandi, M. Rendine, and D. Scuito. Functional verification for SystemC descriptions using constraint solving. In *Design, Automation and Test in Europe (DATE'02)*, pages 744–751, March 2002.

[8] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, MA, 2000.

[9] SystemC Language Working Group. Functional specification for SystemC 2.0.1, April 2002. `http://systemc.org`.

[10] James C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD Thesis, Massachusetts Institute of Technology, June 2000.

[11] James C. Hoe and Arvind. Synthesis of Operation-Centric hardware descriptions. In *International Conference on Computer Aided Design*, pages 511–518. IEEE/ACM, 2000.

[12] S. Liao, S. Tjiang, and R. Gupta. An effieicent implementation of reactivity for modeling hardware in the Scenic design environment. In *Design Automation Conference*, pages 70–75, Anaheim, CA, June 1997.

[13] Xiaowei Shen. *Design and Verification of Adaptive Cache Coherence Protocols*. PhD Thesis, Massachusetts Institute of Technology, February 2000.

[14] Joseph Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In J. N. Oliveira and Pamela Zave, editors, *Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 43–71. Springer-Verlag, 2001.

[15] M.N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *Design, Automation and Test in Europe (DATE'02)*, pages 28–35, March 2002.