Specifying superscalar microprocessors in Hawk

Byron Cook, John Launchbury, and John Matthews
{byron,jl,johnm}@cse.ogi.edu

Oregon Graduate Institute

Abstract. Hawk is a language for the specification of microprocessors at the microarchitectural level. In this paper we use Hawk to specify a modern microarchitecture based on the Intel P6 with features such as speculation, register renaming, and superscalar out-of-order execution. We show that parametric polymorphism, type-classes, higher-order functions, lazy evaluation, and the state monad are key to Hawk's concision and clarity.

1 Introduction

As the performance of cutting edge microprocessors increases, so too does their microarchitectural complexity. For example:

- A superscalar processor that fetches multiple instructions must cache instructions that cannot be immediately executed.
- A processor with out-of-order execution must usually record the original instruction sequence for exception handling.
- A processor that renames registers must allocate and then recycle virtual register names.

While today's hardware description languages (HDLs) suffice for simple microarchitectures, the features of modern designs are difficult to specify without a richer language. Hawk is a specification language based on Haskell [15] that, for the following reasons, provides a strong foundation for a new generation of HDLs:

- Parametric polymorphism allows generic specifications to be used in different contexts.
- Type-classes provide a convenient mechanism for abstracting over instruction sets, register sets, and microarchitectural components.
- Higher-order functions enable a designer to structure specifications in elegant and concise ways.
- Lazy evaluation naturally supports the simulation of multiple mutually dependent streams of instructions and data.
- The state monad facilitates a disciplined style when specifying components with mutable state.

In this paper we explore a Hawk specification of a microarchitecture based on the Intel P6 [4]. We give an overview of the top-level design, and describe in detail our specification of the Reorder Buffer. The purpose of this paper is to show that complex microarchitectures can be formally specified in a clear, concise and intelligible way that facilitates understanding, design review, simulation, and verification.

We assume the reader is familiar with the basic concepts of functional languages and microarchitectural design (such as branch prediction and pipelining). For an in-depth introduction to Haskell, read Hudak, Peterson, and Fasel's tutorial [5]. For more information on microarchitectures, refer to Johnson's textbook [6].

The remainder of this paper is organized as follows: in Section 2 we introduce an architecture; in Section 3, we provide an introduction to Hawk; in Section 4 we use Hawk to specify the architecture; and in Section 5 we highlight how the features of Hawk are used in the specification.

2 A modern microarchitecture

2.1 Machine instruction notation

Throughout this paper we use the following notation for machine instructions:

```
r1 <- r2 + r3
```

The register `r1` is the *destination register* or *destination operand*. Registers `r2` and `r3` are *source registers*.

When the contents of a register is known we may choose to pair the register name and its value:

```
r1 <- (r2,5) + r3
```

In this case, `5` is a *source register value*.

When an instruction's *destination register value* has been computed, we denote this by pairing the destination register with its value:

```
(r1,8) <- (r2,5) + (r3,3)
```

We sometimes refer to a destination register value as the instruction's value.

2.2 Superscalar microarchitectures

In general, superscalar architectures employ aggressive strategies to resolve inter-instruction dependencies and mask the latency of memory accesses. These include speculative execution, the use of virtual register names, and out-of-order instruction issue. The internal microarchitectures often resemble that of a dataflow processor using speculative parallel evaluation. They are thus able to exploit instruction level parallelism to execute sequential, scalar programs.

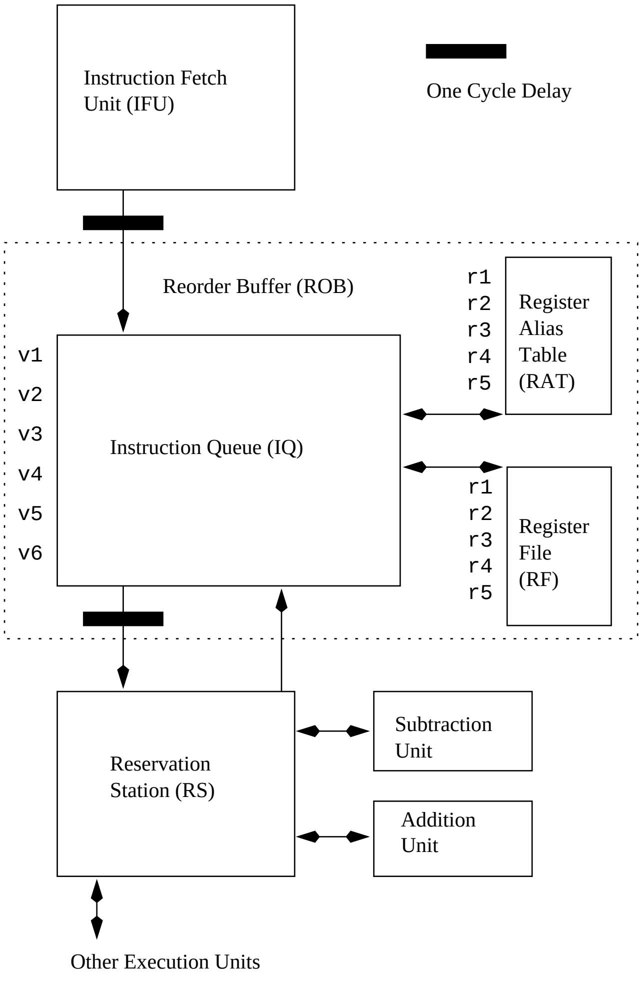

Fig. 1. Microarchitecture

The focus of this paper is on the speculative, superscalar, out-of-order, register renaming microarchitecture shown in Fig. 1. In the remainder of this section we provide an informal introduction to the architecture.

A Reorder Buffer (ROB) maintains the sequential programming model of an architecture while instructions are executed out-of-order and in parallel elsewhere in the processor. In Fig. 1 the ROB is shown as the composite of a circular Instruction Queue, a Register Alias Table, and a Register File for the real register set.

The Instruction Queue (IQ) stores instructions in the order in which they are received from the Instruction Fetch Unit (IFU). The IQ also behaves like a register file for the virtual register set, where the instruction's position in the IQ is its virtual register name.

The Register Alias Table (RAT) is an array of virtual register names indexed by the real register set. For a given real register name, r, the RAT contains either the location of the youngest instruction in the IQ using r as a destination operand; or nothing, if no instruction in the ROB contains the destination operand r. For example, if the instruction `r5 <- r2 + r3` is placed into position `v1` of the IQ (as in Fig. 2), then the real register `r5` is aliased in the RAT to the virtual register `v1`. If `r4 <- r5 + r2` is then inserted into the IQ (Fig. 3), its reference to `r5` is updated to `v1`, and `r4` is aliased to `v2` in the RAT.

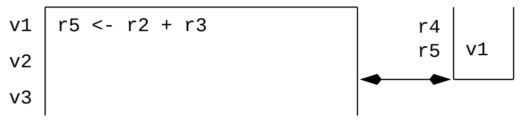

Fig. 2. Inserting `r5 <- r2 + r3` into the ROB

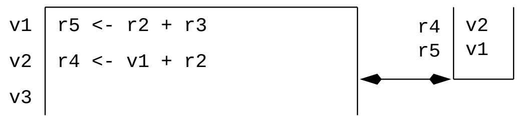

Fig. 3. Inserting `r4 <- r5 + r2` into the ROB

Each instruction, after it has been placed into the ROB, is passed onto the Reservation Station (RS) to be executed. The RS is a data-flow circuit that can execute instructions out-of-order and in parallel. Upon completion in the RS, an instruction's value is returned to the ROB and forwarded to other instructions still in the RS.

2.3 Retiring instructions

An instruction is retired from the ROB when it is at the front of the IQ and its value has been calculated. To retire an instruction in location v with destination operand r, the ROB must write the instruction's value to position r in the Register File, and remove the alias from the RAT if r is still aliased to v.

Why isn't r always aliased to v? Consider the scenario in Fig. 4, where the ROB contains two instructions with `r5` as their destination operand. The virtual register `v1` is no longer an alias of `r5` in the RAT. When retiring the instruction from `v1`, the alias in the `r5` position of the RAT should not be removed. Doing so would remove the unrelated alias from `r5` to `v3`. However, in Fig. 5, because only

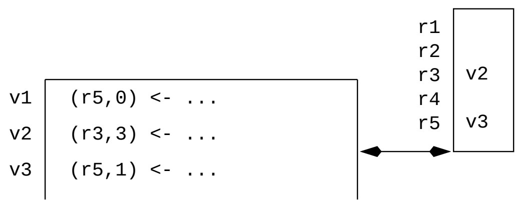

Fig. 4. IQ contains two instructions that alter `r5`

one instruction contains the destination operand `r5`, `r5` remains aliased to `v1`. In this case, when retiring instruction `v1` from the IQ, the alias at the position `r5` in the RAT should be erased.

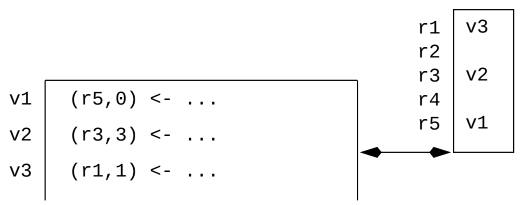

Fig. 5. IQ contains one instruction that alters `r5`

2.4 Example

To illustrate the microarchitecture in action, we trace the execution of a four instruction program:

```
r2 <- r1 + r3
```

```
r4 <- r4 + r2
r2 <- r1 + r1
r1 <- r5 - r3
```

Rather than demonstrating the potential performance of the microarchitecture, this example is tailored to show the amount of bookkeeping that the processor must maintain.

In Fig. 6, execution begins in Cycle 1 with the fetch of four instructions, the last of which requires a different execution unit. In Cycle 2 the fetched instructions are inserted into the IQ. Source register references are modified in one of two ways. Either the operand is replaced with a virtual register reference if it is aliased in the RAT, or the register's value is filled in from the Register File. During Cycle 3 the first and last instructions are executed in parallel. In Cycle 4 the ROB begins retiring instructions based on their position in the instruction sequence. Although the first and last instructions have both completed, to maintain the sequential programming model, only the first instruction can be retired. The last instruction remains in the ROB until its predecessors have all been retired. In Cycle 5, `v2` is computable because the value of `v1` has been forwarded to the source operand. In Cycle 6, because instruction `v2` has completed, the remaining instructions are retired.

3 The Hawk specification language

This section introduces concepts and abstractions used in Hawk. At the risk of incompleteness, we will rely on the reader's intuition to fill in the meanings of functions and syntax that are not described.

3.1 Signals

A signal represents a wire, where at each clock cycle the value of a signal may change. For example, a signal could alternate between `True` and `False`. Or a signal might contain a series of primes numbers. Informally, we can think of a signal as an infinite sequence where the clock cycle is the index:

```
toggle = True, False, True, False, True, False, ....
primes = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,  ....
```

Like the synchronous language Lustre [3], Hawk provides a built-in signal type and functions to construct and manipulate them. The function `constant`, from Fig. 7, returns a signal that does not change over time:

```
constant 5 = 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ....
```

The function `before` delays a signal with a list of initial values[1]:

```
[-1,0] `before` primes = -1, 0, 2, 3, 5, 7, 11, ....
```

[1] `before` denotes that `before` is used as an infix operator

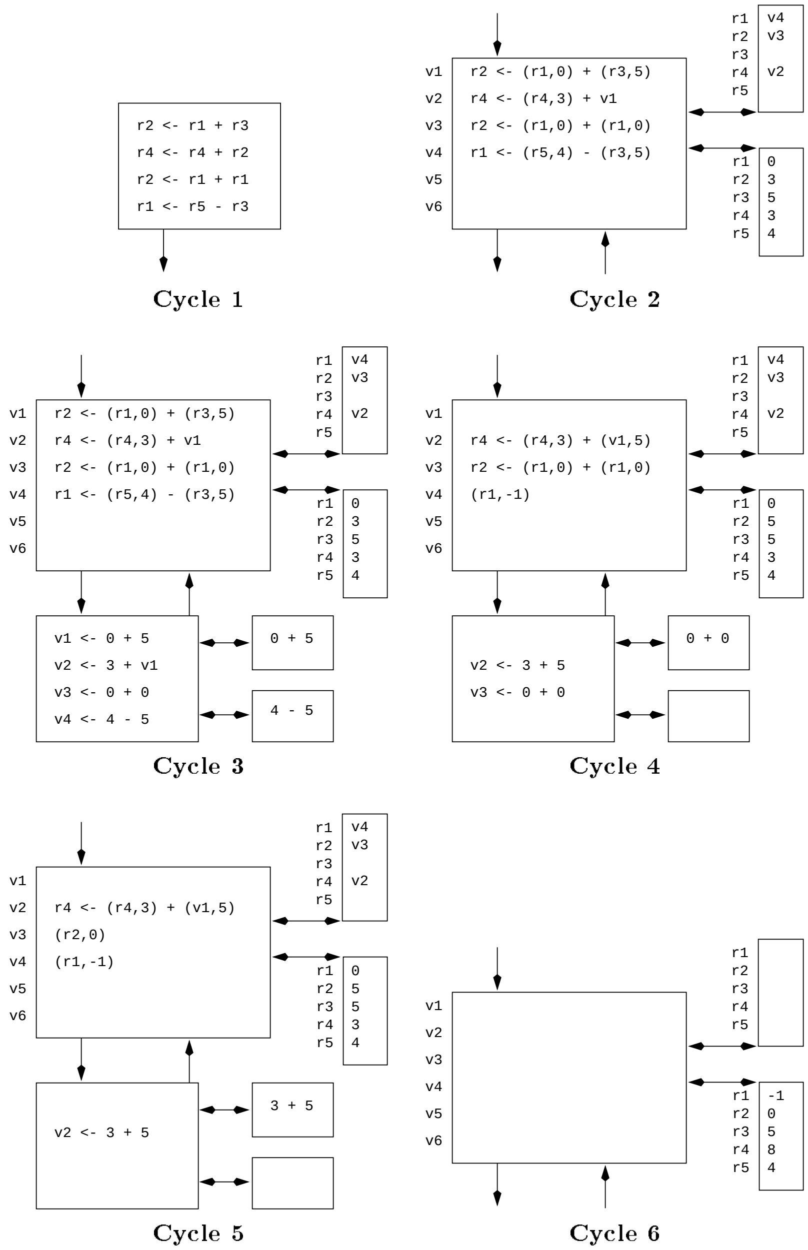

Fig. 6. Example execution trace

```
constant :: a -> Signal a
delay    :: a -> Signal a -> Signal a
before   :: [a] -> Signal a -> Signal a
bundle   :: (Signal a,Signal b) -> Signal (a,b)
unbundle :: Signal (a,b) -> (Signal a, Signal b)
lift     :: (a -> b) -> Signal a -> Signal b
```

Fig. 7. Type signature of primitive `Signal` functions

The function `bundle` takes a pair of signals and returns a signal of pairs:

```
bundle (primes,toggle) = (2,True), (3,False), ....
```

The function `lift` applies its argument to each value in a signal:

```
lift f primes = f 2, f 3, f 5, f 7, f 11, ....
```

Conditional statements are overloaded for signaled expressions. For example:

```
if toggle then primes
          else constant 0    = 2, 0, 5, 0, 11, ....
```

Later in this paper we use the function `delay`, which is defined in terms of `before`:

```
delay x s = [x] `before` s
```

So, for example:

```
delay 6 primes = 6, 2, 3, 5, 7, 11, 13, 17, ....
```

3.2 Transactions

Transactions [1] formalize the notation of instructions introduced in Subsection 2.1. A transaction is a machine instruction grouped together with its state. This state might include:

- Operand values.
- A flag indicating that the instruction has caused an exception.
- A predicted jump target, if the instruction is a branch.
- Other obscure information, such as predicted operand values if we choose to implement value locality [12] optimizations.

Transactions are provided as a library of functions, written in Hawk, for creating and modifying transactions. For example, `bypass` takes two transactions and builds a new transaction where the values from the destination operands of the first transaction are forwarded to the source operands of the second. If `i` is the transaction:

```
(r4,8) <- (r2,4) + (r1,4)
```

and j is the transaction:

```
r10 <- (r4,6) + (r1,4)
```

then `bypass i j` produces the transaction:

```
r10 <- (r4,8) + (r1,4)
```

In our experience, specifications that operate on transactions are more concise than those that treat instructions and state separately. When designed in this style, a processor fetches a transaction containing only the machine instruction which is later refined by the various microarchitectural components until the destination operand value is calculated. Transactions are an example of a user-defined abstraction designed to aid the development of a complex microarchitecture. The concept of an instruction's local state as it acquires its operands, is executed, and finally retired, is the essential concept of a superscalar processor. Transactions also aid the verification process because they make explicit much of the state needed to prove correctness. In lower-level specifications this data has to be inferred from the context.

4 Specifying the microarchitecture

Fig. 8 contains the top-level Hawk specification of the microarchitecture in Fig. 1. Using lazy evaluation, a Hawk simulation will solve the specification's system of mutually dependent equations, producing a computational simulation. The components of the microprocessor are modeled as functions from input signals to output signals. For example, as Fig. 9 illustrates, the ROB is a component with two inputs and four outputs. The inputs and outputs may each represent very wide connections — perhaps enough to move numerous transactions in a single cycle. The arguments and results of the function `rob` from Fig. 8,

```
(retired,ready,n,err) = rob 6 (fetched,computed)
```

except for the size parameter, correspond to those in Fig. 9.

4.1 Top-level structural specification

In Fig. 8 the first equation specifies how transactions are fetched from the instruction memory, `mem`:

```
(instrs,npc) = ifu 5 mem pc err ([5,5] `before` n)
```

The Instruction Fetch (IFU) function, `ifu`, uses its first parameter, `5`, to determine the maximum number of transactions to fetch at each cycle. The IFU retrieves consecutive transactions beginning at the program counter, `pc`. Initially, during the first and second cycles, 5 transactions are fetched. In later cycles feedback from the ROB, `n`, is used to determine the number of transactions to fetch.

Execution begins with the transaction at location `256` in the instruction memory. After the first cycle, the value of `pc` depends on the location of the

```
processor mem = retired
 where

 (instrs,npc) = ifu 5 mem pc err ([5,5] `before` n)

 pc = delay 256 (if err then lastpc retired else npc)

 fetched = delay [] (annotate instrs)

 (retired,ready,n,err) = rob 6 (fetched, computed)

 computed = rs (6,execUnits) (delay False err,delay [] ready)

 memU = mob fetched retired

 execUnits = [addU,subU,jmpU,intU,fltU,memU]
```

Fig. 8. Top-level microprocessor specification

previously fetched transaction, and the possibility of a mispredicted branch or exception. In the event of a mispredicted branch or exception, the signal `err` is set, and the `pc` comes from the last retired transaction:

```
pc = delay 256 (if err then lastpc retired else npc)
```

For simplicity we employ a naive branch prediction algorithm — all branch transactions are simply assumed to jump to the next consecutive transaction. The function `annotate` places this guess into the state of branch transactions:

```
fetched = delay [] (annotate instrs)
```

The Reservation Station (RS) function, `rs`, is parameterized on its size and execution units:

```
computed = rs (6,execUnits) (delay False err,delay [] ready)
```

During the initialization of `rs`, the execution units are clustered together with a function. The execution units can be pipelined or blocking. Execution units can also complete in multiple clocks. The RS accepts two input signals: an error flag and transactions from the ROB. The transactions `computed` contains the transactions that are complete and ready to be updated in the ROB.

4.2 ROB specification

Whereas the top-level specification of the microarchitecture is easily constructed as a purely functional application of components, the ROB is more complicated. Certainly the ROB could be specified in the applicative style used in Fig. 8. However, at a higher level of abstraction, the ROB can be thought of as a circuit

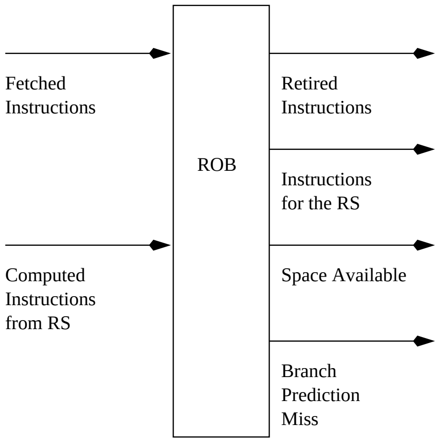

Fig. 9. Inputs and outputs of the ROB

that sequences destructive updates on mutable components. Our approach in this paper is to specify the ROB in a behavioral style using imperative language features. In Fig. 10, the specification of the ROB is provided in the state monad and then encapsulated with Hawk's state thread encapsulation construct `runST` [9]. The advantage of using `runST` is that the language guarantees that `rob` neither depends on nor alters mutable state in other components or an outside environment [10]. We can therefore treat the ROB as a pure function that, on a given input, always returns the same output.

In Fig. 10, during the beginning of the simulation, the ROB constructs its mutable sub-components (much of this work would be fabricated into the processor):

```
q <- IQ.new n
rat <- RAT.new
rf <- RF.new
```

At each cycle the ROB takes the `fetched` and `computed` signals signals

```
cycle(fetched,computed)
```

and performs the following tasks:

- Update the `computed` transactions in the queue. For each transaction in the `computed` list, the function `update` obtains the virtual register reference from the destination register, and uses it as the index when updating the queue:

  ```
  update q computed
  ```

- Insert the `fetched` transactions into the queue (see Subsection 4.3):

  ```
  instrs <- insert rat q rf fetched
  ```

- Find transactions from the front of the queue that are ready to be retired. If a retired transaction was a mispredicted branch or raised an exception, then only retire the transactions before it (see Subsection 4.4):

```
(retired,err) <- retire rat q rf
```

- If a retired transaction was a mispredicted branch or raised an exception, then clear the IQ and RAT:

```
if err then do {q.clear; rat.clear}
```

- Measure the capacity of the queue for the IFU:

```
capacity <- q.space
```

- If a retired transaction was mispredicted or raised an exception then do not send `fetched` transactions to the RS:

```
let ready = if err then [] else instrs
```

- Return the retired transactions, the transactions ready to pass onto the RS, the measured capacity, and the error flag:

```
return (retired,ready,capacity,err)
```

```
rob n (fetched,computed)
  = runST (
    do { q <- IQ.new n
       ; rat <- RAT.new
       ; rf <- RF.new
       ; cycle(fetched,computed)
            { update q computed
            ; instrs <- insert rat q rf fetched
            ; (retired,err) <- retire rat q rf
            ; if err then do {q.clear; rat.clear}
            ; capacity <- q.space
            ; let ready = if err then [] else instrs
            ; return (retired,ready,capacity,err)
            }
       }
    )
```

Fig. 10. ROB behavioral specification

```
insert rat q rf instrs
  = foreach t in instrs
    do { (reg,alias) <- q.assignAddr (head (getDestRegs t))
       ; src <- mapM (rat.replace) (getSource t)
       ; rat.write reg alias
       ; dest <- mapM (rat.replace) (getDest t)
       ; new <- regRead q rf (trans dest (getOp t) src)
       ; q.enQueue new
       ; return new
       }
```

Fig. 11. Insertion specification

4.3 Inserting new instructions

Fig. 11 contains the specification of the function `insert`. When inserting new transactions into the ROB, `insert` takes a list of transactions, `instrs`, and performs the following actions:

- Calculate the new position in the queue for the transaction:

  ```
  (reg,alias) <- q.assignAddr (head (getDestRegs t))
  ```

- Substitute references to real registers with virtual registers in the source operands:

  ```
  src <- mapM (rat.replace) (getSource t)
  ```

- Update the RAT:

  ```
  rat.write reg alias
  ```

- Substitute the reference from the real destination register to the virtual destination register:

  ```
  dest <- mapM (rat.replace) (getDest t)
  ```

- Read real register references:

  ```
  new <- regRead q rf (trans dest op src)
  ```

- Enqueue the transactions:

  ```
  q.enQueue new
  ```

- Return the updated transactions:

  ```
  return new
  ```

```
retire rat q rf
  = do { perhaps <- q.deQueueWhile complete
       ; let (retired,err) = hazard findErr perhaps
       ; mapM (writeOut rf rat) retired
       ; return (retired,err)
       }
  where findErr t = jmpMiss o exceptionRaised

jmpMiss t = do { x <- getPC t
               ; y <- getSpecPC t
               ; return (x /= y)
               }
            `catchEx` False

writeOut rf rat t =
 do { let [Reg (Virtual vr real) (Val x)] = getDest t
    ; rf.write real x
    ; a <- rat.read real
    ; do { v <- a ; guard (v == vr) ; return (rat.remove real) }
      `catchEx` return ()
    }
```

Fig. 12. Retirement specification

4.4 Retiring instructions

Fig. 12 contains the specification of the function `retire`. When retiring transactions from the ROB, `retire` performs the following actions:

- Remove transactions from the front of the queue until a transaction is found that has not been computed:

```
perhaps <- q.deQueueWhile complete
```

- If a branch was mispredicted or an exception was raised then ignore all of the transactions after that transaction:

```
let (retired,err) = hazard findErr perhaps
```

- Write the values of the destination registers to the Register File :

```
mapM (writeOut rf rat) retired
```

- Return the retired transactions and a flag indicating a branch miss or raised exception:

```
return (retired,err)
```

5 Conclusions

The design of correct superscalar microarchitectures is difficult. The language of discourse must be powerful enough to describe a wide range of processors, and concise enough that designers can maintain intellectual control of their work. Moreover, the language must scale to the designs of the future. In this section we highlight how polymorphism, type-classes, higher-order functions, lazy evaluation and the state monad improve the concision, clarity, and perhaps the provability of our specification.

5.1 Polymorphism

Many of Hawk's library functions are polymorphic. For example, `delay` accepts an argument of type `a` (where `a` could be any type), a signal of `a`, and returns a new signal of `a`:

```
delay :: a -> Signal a -> Signal a
```

In Fig. 8, `delay` is used on both Booleans and lists of transactions:

```
(delay False error, delay [] ready)
```

Without parametric polymorphism, a delay function would be required for each specific type. In many specification languages, because the types that can be passed through signals are limited, ad hoc solutions are usually sufficient. However, signals in Hawk are unrestricted and therefore must be accompanied by truly polymorphic functions.

5.2 Type-classes

The RAT, used in Fig. 10, is abstracted over the register set used in the underlying machine language. For example, the function `RAT.new` is of type:

```
RAT.new :: Register r => ST s (RAT s r v)
```

This reads "for any type `r` that is a register set, `RAT.new` constructs a new RAT indexable by `r`". Because `r` is an instance of `Register`, the variables `minBound` and `maxBound` are overloaded to the smallest and largest values of `r`:

```
minBound :: Register r => r
maxBound :: Register r => r
```

`RAT.new` uses `minBound` and `maxBound` to determine the size of the constructed RAT.

Without type-classes, the RAT would either be useful for only one particular register type, or a number of extra parameters (such as the bounds and comparison functions) would need to be passed to the functions `rob`, `RAT.new`, `insert`, etc. Type-classes allow us to easily adapt the RAT to other machine languages, such as IA-64 or PowerPC.

5.3 Higher-order functions

Higher-order functions allow designs to be parameterized in new and powerful ways. For example, in Fig. 8 the RS is parameterized over a list of execution units. At the start of a simulation, the RS builds a single execution unit by clustering the list of circuits. When testing various microarchitectural configurations, the designer can easily modify the execution units at the top-level.

We might also want to abstract the RS on the scheduling function:

```
computed = rs (6,cluster,[addU,subU,jmpU,mltU])
              (delay False error,delay [] ready)
```

This way we might use the same RS specification in many instantiations with different configurations of scheduling functions and execution units.

5.4 Lazy evaluation

Without Hawk's lazy semantics we would not be able to write the dependent equations in Fig. 8. Consider the simple clock circuit in Fig. 13. The design is

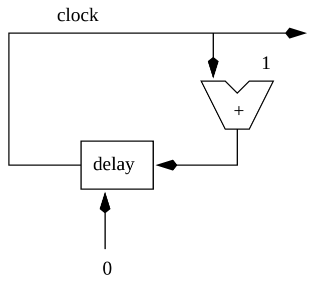

Fig. 13. Clock circuit

easily specified as a Hawk expression where the value depends on itself:

```
clock = delay 0 (clock + 1)
```

In a strict semantics, the meaning of this expression would be undefined.

5.5 Encapsulated state

While maintaining the mathematically consistent features of Hawk, such as polymorphism and lazy evaluation, the state monad adds the ability to use mutable state directly rather than encoding state with delays and other lower level signal constructs. The use of `runST` facilitates the safe integration of imperative specifications in an applicative framework.

6 Future work

Currently, using the Glasgow Haskell Compiler, the simulator derived from the specification in this paper retires 800 instructions per second when executed on a UltraSPARC-1 processor. We hope that to improve performance using domain specific optimizations or compilation to better simulation packages.

We have not sufficiently explored the synthesis and analysis of Hawk specifications. Although Hawk is at a higher level of abstraction than mainstream HDLs from our initial results we believe that, within limits, automatic synthesis is feasible.

We have just completed a correctness proof of a microarchitecture based on this paper in which the ROB, RS, and IFU are specified axiomatically [8]. We now hope to prove that the ROB, RS, and IFU presented here implement the axioms.

We hope to use Hawk formally to verify the correctness of microprocessors with a mechanical theorem prover (for example, Isabelle [14]). A theorem proving environment for Hawk must have support for manipulating higher-order functions and polymorphic types.

7 Related work

Ruby [7] is a specification language based on relations, rather than functions. Relations can describe more circuits than functions. Much of Ruby's emphasis is on circuit layout. Ruby provides combinators to specify where circuits are located in relation to each other and to external wires. Hawk's emphasis is on circuit correctness, so we do not address layout issues.

Lava is a Haskell library for the specification of Field Programmable Gate Arrays. Lava is intended to be used at a lower level of abstraction than Hawk. Like Ruby, Lava specifications focus much attention on issues related to layout.

Like Hawk, Lustre [3] and the other reactive synchronous languages (Signal, Esterel, Argos, etc) provide mechanisms for constructing expressions over time-varying domains. However, research on these languages has emphasised reactive features rather than the issues addressed in this paper.

The Haskell library Hydra [13] allows modeling of gates at several levels of abstraction, ranging from implementations of gates using CMOS and NMOS pass-transistors, up to abstract gate representations using lazy lists to denote time-varying values. Hydra is similar to Hawk in many respects. However composite signal types, such as signals of integers, must be constructed as tuples or lists of Boolean signals. This restriction severely limits Hydra's application to the domain of complex microarchitectures.

HML [11] is a hardware modelling language based on ML. It supports higher-order functions and polymorphic types, allowing many of the same abstraction techniques that are used in Hawk. On the other hand, HML is not lazy, so it does not easily allow the dependent circuit specifications that are key in specifying microarchitectures in Hawk. Also, HML does not clearly separate its imperative and functional features.

MHDL [2] is a hardware description language for describing analog microwave circuits, and includes an interface to VHDL. Though it tackles a very different area of the hardware design spectrum, like Hawk, MHDL is essentially an extended version of Haskell. The MHDL extensions have to do with physical units on numbers, and universal variables to track frequency, time, etc.

8 Acknowledgements

For their contributions to this research, we thank Borislav Agapiev, Mark Aagaard, John O'Leary, Robert Jones, Todd Austin, and Carl Seger of Intel Corporation; Elias Sinderson and Neil Nelson of The Evergreen State College; Dick Kieburtz, Jeff Lewis, Sava Krstić, Walid Taha, and Andrew Tolmach of the Oregon Graduate Institute; Simon Peyton-Jones of the University of Glasgow; and the members of BWERT.

This research was supported by Intel and Air Force Material Command (F19628-93-C-0069). John Matthews is supported by a fellowship from the National Science Foundation.

Note: This paper appears in the proceedings of the 1998 Workshop on Formal Techniques for Hardware (Marstrand, Sweden)

References

1. Aagaard, M., and Leeser, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).
2. Barton, D. Advanced modeling features of MHDL. In *International Conference on Electronic Hardware Description Languages* (Jan. 1995).
3. Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages* (Munich, Germany, Jan. 1987).
4. Gwennap, L. Intel's P6 uses decoupled superscalar design. *Microprocessor Report 9*, 2 (1995).
5. Hudak, P., Peterson, J., and Fasel, J. A gentle introduction to Haskell. Available at www.haskell.org, Dec. 1997.
6. Johnson, M. *Superscalar Microprocessor Design.* Prentice Hall, 1991.
7. Jones, G., and Sheeran, M. Circuit design in Ruby. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North-Holland, 1990.
8. Krstić, S., Cook, B., Launchbury, J., and Matthews, J. A correctness proof of a speculative, superscalar, out-of-order, renaming micro-architecture. Submitted to 1998 Formal Methods in Computer Aided Design, Apr. 1998.
9. Launchbury, J., and Jones, S. P. Lazy functional state threads. In *Programming Languages Design and Implementation* (Orlando, Florida, 1994), ACM Press.
10. Launchbury, J., and Sabry, A. Monadic state: Axiomatization and type safety. In *International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

11. Li, Y., and Leeser, M. HML: An innovative hardware design language and its translation to VHDL. In *Conference on Hardware Design Languages* (June 1995).
12. Lipasti, M. H. *Value Locality and Speculative Execution.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1997.
13. O'Donnell, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).
14. Paulson, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.
15. Peterson, J., and et al. Report on the programming language Haskell: A non-strict, purely functional language, version 1.4. Available at www.haskell.org, Apr. 1997.