Extensible Control Architectures

Greg Hoover
University of California, Santa Barbara
Engineering I
Santa Barbara, California 93106
ghoover@ece.ucsb.edu

Forrest Brewer
University of California, Santa Barbara
Engineering I
Santa Barbara, California 93106
forrest@ece.ucsb.edu

Timothy Sherwood
University of California, Santa Barbara
Engineering I
Santa Barbara, California 93106
sherwood@cs.ucsb.edu

ABSTRACT

Architectural advances of modern systems has often been at odds with control complexity, requiring significant effort in both design and verification. This is particularly true for sequential controllers, where machine complexity can quickly surpass designer ability. Traditional solutions to this problem require elaborate specifications that are difficult to maintain and extend. Further, the logic generated from these specifications bares no resemblance to the intended behavior and often fails to meet design performance constraints. In the process of designing a multi-threaded, dynamically-pipelined microcontroller, we encountered a number of common difficulties that arise from the inadequacies of traditional pipeline design methodologies. Through the use of a novel nondeterministic finite automata (NFA) specification model, we were able to implement an extensible control structure with minimal design effort. In this paper we present a viable pipeline controller specification methodology using the pyPBS language, which enables minimal effort control partitioning and compact behavioral representation. The structure of the language encourages design decisions that promote efficient modular constructions which can be easily integrated and extended. We present an overview of the our methodology including background on the pyPBS synthesis model, an architectural overview of our multi-threaded microcontroller, and implementation details for the control structure of the design including the complete control specifications. In addition, we show that the applicative nature of the pyPBS language allows for addition of a multi-cycle multiplication unit with minimal effort.

Categories and Subject Descriptors

B.1.1 [**Hardware**]: Control Structures and Microprogramming—*hardwired control*; B.1.2 [**Hardware**]: Control Structures and Microprogramming—*automatic synthesis*; B.6.1 [**Hardware**]: Logic Design—*sequential circuits*; B.6.3 [**Hardware**]: Logic Design—*automatic synthesis, hardware description languages*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'06, October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-XXX-X/06/0010 ...$5.00.

General Terms

Design, Languages

Keywords

control architecture, specification methodology

1. INTRODUCTION

Many embedded systems require a single processor to serve a broad array of functions such as operating interrupt driven devices, managing wireless networks, processing software updates, and performing distributed computations. As part of our ongoing research we are examining the usefulness of architectural techniques in simplifying the design of software for such systems. In particular, hardware support for multi-threading can reduce code footprint for save/restore overhead, it can enable zero-cycle interrupt response from service threads, and potentially even reduce software complexity. Despite these advantages, embedded support for multi-threading hardware will only ever become a reality if it can be introduced without significantly increasing the control complexity of the underlying hardware.

Embedded processors, unlike mainstream microprocessors, are extremely sensitive to increased control complexity because the design and verification teams must be incredibly small, and the designs need to be both flexible and extensible. For a small 2-stage pipeline, the control space for a single threaded design can be fully enumerated. However, with a 6-stage pipeline, multi-cycle stalls, interrupts, exceptions, and any possible interleaving of up to 8 threads, the control state space quickly becomes unmanageable. Even if a standard state chart based approach could be constructed, the lack of control compartmentalization would force a reevaluation of the entire control state space to make even a simple modification.

Standard design methods, based on finite state machine based control synthesis, have the very undesirable property that combining n machines of size k will result in a machine of size up to k^n. This model assumes that all parts of the machine state need to be uniformly accessibly. This inefficiency means that it falls upon the designers to manually partition their control structures and to ensure that every possible combination of states either leads to correct execution or is unreachable. Because of this, designing modern control systems is typically an ad-hoc process specifically suited to a particular application.

Consider the specification of a five stage interlocked pipeline in a HDL. Such a design would typically begin with de-

terministic modeling of intended pipeline behavior. While well-suited for machines with a single active point of control, deterministic models expand rapidly when multiple active points of control can occur simultaneously, such as in pipelines. Thus the deterministic state space for our pipelined processor constitutes the language containment (or closure) of a non-pipelined implementation. Although specification of this model is not impractical, it is clear that complexity grows quickly and the job is not made any easier by the sequential logic constructs of a conventional HDL. Because of the mixing of control sequences and functional actions, specifications in a conventional HDL span multiple pages of source code. This makes it very difficult to recover the overall behavioral meaning of the specification and requires modifications at various locales for even simple control changes, such as adding a new stall condition.

Instead, our compartmentalized control method allows the HDL specification to be dissected into a hierarchy of cleanly interacting controllers, where each has the ability to stall or flush the entire pipeline. Such partitioning naturally arises from the methodology, where a common interlocked pipeline controller is initially defined, creating a reliable control infrastructure from which additional behavior can be incrementally extended. The use of the pyPBS language means that such additions are additive, where the scope of design changes are limited, as are the resulting circuit-level changes. Drawing from nondeterministic finite automata (NFA) technology, pyPBS provides inherent support for systems having multiple active points of control (they can be in multiple "states" at once). Such behavior is naturally supported by the parallel nature of hardware, making NFA-based synthesis well suited for describing the behavior of concurrent pipelines. Better still, the control is specified in a modular and intuitive way, and the description of a pipeline controller actually resembles a pipeline.

Specifically, we make the following contributions:

- We propose a novel method for creating extensible and compartmentalized pipeline control architectures based on the incremental addition of stall and flush recognizers.

- We show that by embracing an NFA-based approach, the complexity inherent in specifying pipelines can be naturally expressed and synthesized even for complex instruction sets.

- To demonstrate the usefulness of our methods, we have designed and implemented an 8-bit 6-stage 8-way multithreaded microcontroller fully compatible with the Atmel AVR instructions set. We show that the complex control required for this design can be easily extended from a basic interlocked pipeline specification.

- We further show such specifications to be easily extended through the addition of 3 different types of multiply units.

To show that our design method is not simply a theoretical exercise, we have designed and implemented a family of 8-bit instruction-interleaved microcontrollers based on the Atmel AVR instruction set. The remainder of this paper describes our novel control specification solution, and its use in the design and implementation of a synthesizable multithreaded microcontroller. An overview of pyPBS and its synthesis method are discussed in Section 2. Section 3 describes the incremental design methodology and the fundamental pipeline construction that serves as the design starting point. Section 4 presents an architectural overview of the design, with control details including pyPBS specification and the resulting gate-level logic given in Section 5. The design is shown to be easily extended to include a multi-cycle multiplier in Section 6. Related work pertaining to the state of the art in high-level specification is presented in Section 7. Finally, concluding remarks are given in Section 8.

2. PYPBS

The rationale behind pyPBS stems from the practical issues of constrained hardware design where a designer commonly works with a rather restrictive set of sequential constraints – memory protocols, interfaces, and previously constructed designs. In such designs, it is desirable to have an applicative design methodology that allows incremental integration of system components. pyPBS provides such a design framework through its simple context-based execution model. The semantics of generated machines follow from a non-deterministic finite automaton (NFA) model where execution is assumed to be as general as possible. The notion of context corresponds to an active point of control in the automaton model. Rather than provide coherence by restricting the legal execution paths of the specification, all paths execute in a completely general fashion. Thus the resulting machine can have many active points of control, making it a natural fit for pipelining [9].

The generality of the context model naturally fits with a one-hot encoding scheme, where the acceptance or state of a control point can be represented by the value of a bit. While such machine encodings are typically register-heavy, they prove to be well suited for high performance designs and not uncharacteristically large for designs generated from high level languages. This observation is a consequence of the sparsity of the machine encoding which typically results in simple control logic and offers unique opportunities for optimization. The locality of this encoding technique further provides a system in which synthesis is applicative and the scope of design circuit changes are limited for corresponding changes in the control hierarchy.

Drawing from the expressiveness of extended regular expressions, the pyPBS language enables compact specifications that can be exponentially smaller than their conventionally specified counterparts. Through the addition of a powerful set of language constructs, these extended regular expressions can be used to define complex control sequences in a single line of code. Further, the behavior of such sequences can be understood without reconstructing state charts or scrolling through many lines of code, as is necessary in conventional HDL. Despite their efficiency, regular expressions tend to be rather cryptic, resulting in poor readability. For this reason, pyPBS organizes regular expression fragments inside of a production-based language. This format allows for structured specifications that reuse common control fragments.

Figure 1 depicts the desired behavior for a simple memory write controller. On each cycle, the controller should be able to accept a request and subsequently generate outputs for latching operands, writing to the bus, and releasing the bus. Such behavior requires minimally three states as shown in the 'memory' production of Figure 2. Busses com-

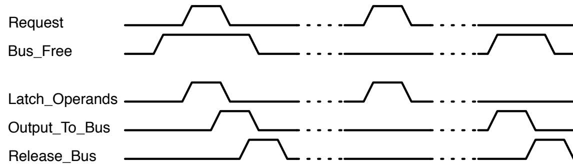

Figure 1: Waveform illustrating the desired behavior of a simple memory write controller.

monly share access between multiple components, and as such, require some type of arbitration. This functionality is provided in the pyPBS specification through insertion of a wait state throttled by an input signal signifying bus availability. Figure 3 depicts the synthesis process beginning with construction of abstract syntax trees (AST) for each pyPBS production. The ASTs are subsequently combined to form a single production directed acyclic graph (DAG) [13]. By using binary decision diagrams (BDD) to represent the logic at each node, a circuit can be directly extracted from the production DAG. The generated machine is output in Verilog HDL and can be directly synthesized or integrated into larger designs [9]. The absence of any functional component specification in this example alludes to the separation of control and data path specifications. Intermixing functional and control specifications reduces readability and makes, even simple changes, difficult to perform – modification to conventional specifications typically require changes at multiple locales, even for basic changes.

3. INCREMENTAL DESIGN METHODOLOGY

In any design flow, it is critical to make use of existing components to reduce development and verification time. Our methodology does just this, using a proven interlocked pipeline specification from which custom behavior can be incrementally added. The pyPBS specification for a 6-stage interlocked pipeline is presented in Figure 4. It should be clear from the symmetry of the specification that the number of pipeline cycles can be changed very easily. While this specification can appear cryptic to the uninitiated, its behavior is quite simple. The control entry point is the beginning of the 'control' production. In each cycle, control is passed to the 'cycle' production which describes the sequential flow of the pipeline – the comma (',') operator signifies sequential concatenation. Each of the pipeline stages is subsequently described on an individual basis, allowing for catering of stage behavior through addition of custom recognizers.

In a tail-throttled pipeline, a stage stalls if it can execute (accept) in the current cycle and its successor stage is stalled. This behavior is specified using a one-or-more ('+') expression, effectively creating a control loop that terminates only upon termination of stall conditions. In this simple case, these stall conditions are generated solely by successor stages and referred to by name – reference names appear between carrots ('<>'). The top row of flip-flops in Figure 5 depicts the resulting stall logic chain and supporting bits. The bottom row similarly represents supporting logic for stage acceptance. This behavior is easily defined in the specification as the sequential complement ('~') of the respective stall condition. From Figure 5, it can be seen that each stage accepts if its predecessor stage has completed and its successor stage is not stalled. During a stall, an active point of control is maintained in the respective stall bit for use upon termination of any stall. In short, each stage monitors two inputs from its respective predecessor and successor stages, stalling if its predecessor did work in the previous cycle and its successor stage is stalled. Subsequent termination of the stall condition allows the stage to execute, handing off control to its successor in the next cycle.

High-level specifications typically bare no resemblance to the circuits which they generate, however, pyPBS reflects specification simplicity in its output. This novel synthesis language draws from a simple, yet flexible NFA execution model which allows the designer to incrementally extend the basic pipeline specification through addition of specialized recognizers. Whereas such modifications would typically be a major undertaking in a conventional HDL, they can be done in pyPBS with little effort. This is in part due to separation of control and data path specifications – pyPBS targets control specification with separate data path specification via any number of techniques. Further, synthesizable Verilog HDL output provides a mechanism for simple integration in any design flow.

4. PROCESSOR ARCHITECTURE

To demonstrate our design methodology we have implemented JackKnife, a multi-threaded, dynamic pipelined microcontroller family that models its architecture after that of the popular Atmel AVR, providing advanced capabilities while maintaining simplicity and software compatibility. Drawing from the memory-mapped design of its progenitor, JackKnife provides uniform interfaces to system peripherals and memory, reducing design complexity. Often the environments in which microcontrollers of this scale are deployed are event-driven in nature, where control flow changes often and unexpectedly. Successful systems in this domain should minimize the overhead of these transitions. The original AVR architecture utilizes a two stage dynamic pipeline which executes instructions in anywhere from one to four cycles, depending on instruction complexity. Single-cycle operation greatly reduces design complexity but results in implementations that top out at 20MHz. JackKnife inherits the same dynamic pipeline behavior, but extends the pipeline to six stages and provides single-cycle support for a much greater number of instructions than the AVR. Dynamic pipelining comes at the expense of added complexity – most notably support for stalling and flushing of pipeline stages.

Multi-threading provides a mechanism for hiding instruction latency in our elongated pipeline. It has the additional benefits of providing low-latency interrupts and zero-cycle context switching, two features which are aptly suited for embedded applications. The interleaved scheduler provides implicit data path sharing and removes data dependencies between consecutive instructions, alleviating many stall conditions and providing better utilization of processor resources. Dynamic interleaving allows the scheduler to dynamically select threads for execution based on resource availability. This provides better processor utilization and system flexibility by bypassing inactive or stalled threads. The JackKnife architecture focuses on flexibility of design and system responsiveness, selecting active threads on a

```
machine memory_write_controller {
        input Request, Bus_Free;
        output Latch_Operands, Output_To_Bus, Release_Bus;

        memory -> .* , wait , write , cleanup;
        wait -> Request <Latch_Operands>;
        write -> ~Bus_Free* , Bus_Free <Output_To_Bus>;
        cleanup -> . <Release_Bus>;
}
```

Figure 2: pyPBS specification for a simple memory write controller supporting bus arbitration.

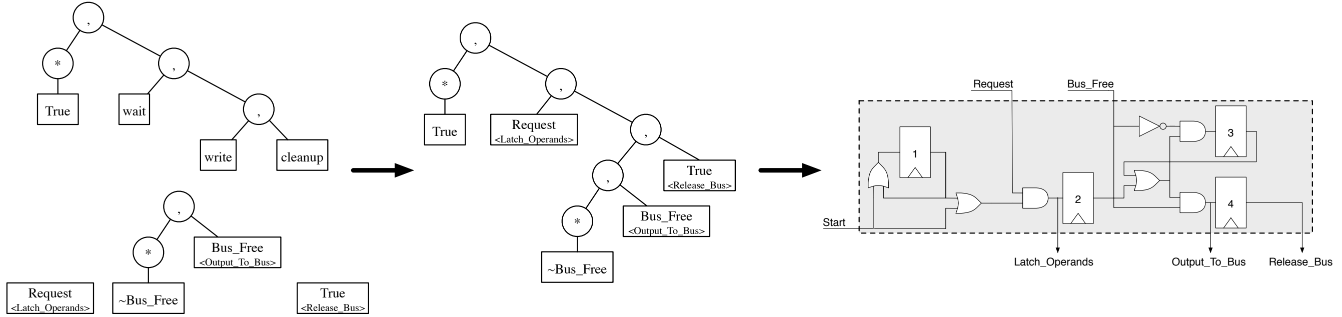

Figure 3: Synthesis process for the simple memory write controller depicting the representative production ASTs, merged production DAG, and resulting circuit.

round-robin basis and scheduling interrupt services threads (IST) on the next cycle.

JackKnife supports concurrent execution of any number of threads up to the current design limit of eight. Interleaved processors often suffer from performance degradation when the number of executing threads drops below some threshold[15]. We have escaped this drawback through the selective addition of operand forwarding. Forwarding logic in the register file typically reduces the length of any data dependency stall to a single cycle. The combination of instruction throughput, low-latency interrupts, and dynamic thread scheduling allows JackKnife to respond to randomness exhibited in many embedded environments. For an in-depth look at the processor architecture and impacts of multi-threading on embedded systems is available in [10].

5. CONTROL ARCHITECTURE AND SPECIFICATION

The addition of pipelining and multi-threading to a single-cycle processor design would prove a non-trivial task if not for the use of pyPBS control specification techniques. Enabling multi-threading comes at only minor expense in terms of modifications to the register file, status registers, and stack pointer registers. However, more significant additions in the way of a scheduler and modifications to the control logic are necessary. Managing the control logic, and all of the states needed to handle the interactions of data path stalls, flushes, interrupts, and multiple threads, can get quickly out of hand. One of the problems is that the semantics of an instruction stream should hold no matter how it is interleaved with streams from other threads. A typical state machine approach to building such a controller can quickly explode into an unmanageable number of states. The other option is to hand build a control structure that can effectively handle a particular instance of the design. This, however, is against our design goal of creating a configurable and extensible design. For instance, it would make changing the number of hardware level threads handled by the architecture very difficult.

To cope with these issues, our methodology begins with the basic pipeline described earlier, extending functionality through simple addition of recognizers. pyPBS provides simple methods for adding custom stall and flush conditions, greatly simplifying the design process and giving way to straightforward implementations for the remaining control units. The resulting controller is shared between all hardware threads, providing a single interlocked structure on which to build the intricacies of the system.

Maintaining machine semantics for simultaneous execution of multiple instruction streams in a common data path requires the additional ability to match instructions with their respective streams. This matching facilitates context selection for retrieval and writeback of instruction operands, identification of stall and flush conditions, control of forwarding mechanisms, and thread scheduling. While per-thread controllers could be used to disambiguate pipeline execution within the pyPBS specification, JackKnife implements a simpler solution, tagging instructions at all stages of the pipeline. This mechanism allows both global and local tag comparisons for various control elements. While it is possible to design a single monolithic controller for this system, the specification and sustainability of such an approach is less than desirable. For this reason, the control structure for JackKnife is partitioned into three controllers, the hierarchy of which is discussed in Section 5.1.

```
machine pipeline_controller {
        input Stall;

        output Stage_1, Stage_2, Stage_3, Stage_4, Stage_5, Stage_6;
        output Stall_Stage_1, Stall_Stage_2, Stall_Stage_3,
               Stall_Stage_4, Stall_Stage_5, Stall_Stage_6;


        control ->  .* , cycle;
        cycle ->    stage_1 , stage_2 , stage_3 , stage_4 , stage_5 , stage_6;

        stage_1 -> (~(Stall_Stage_2 <Stall_Stage_1>+)) <Stage_1>;
        stage_2 -> (~(Stall_Stage_3 <Stall_Stage_2>+)) <Stage_2>;
        stage_3 -> (~(Stall_Stage_4 <Stall_Stage_3>+)) <Stage_3>;
        stage_4 -> (~(Stall_Stage_5 <Stall_Stage_4>+)) <Stage_4>;
        stage_5 -> (~(Stall_Stage_6 <Stall_Stage_5>+)) <Stage_5>;
        stage_6 -> (~(Stall <Stall_Stage_6>+)) <Stage_6>;
}
```

Figure 4: pyPBS specification for the basic 6-stage interlocked (tail-throttled) pipeline controller.

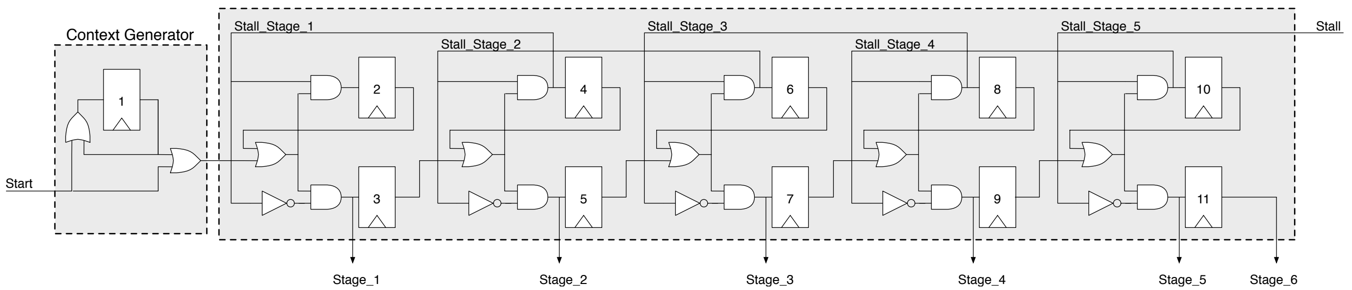

Figure 5: Generated circuit for the basic 6-stage interlocked (tail-throttled) pipeline controller.

5.1 Control Hierarchy

Design partitioning is essential to any sustainable design. While the JackKnife design is comprised of standard function components, the interconnection and control of these components is anything but standard. Integrating the diversity of the instruction set into a single controller has proven difficult in past implementations. Control partitioning allows the control for each functional unit to be specified and verified individually. The simple context-based model of pyPBS facilitates communication between controllers via two simple signals: 'request' and 'stall'.

The control hierarchy is depicted in Figure 6. At the top of this hierarchy is the pipeline controller which provides interlocking (tail-throttled) and support for per-stage stall and flush. This controller operates a dynamic pipeline consisting of six stages: schedule, fetch, decode, read, execute, and commit. It uses abstracted signals to effect major control changes in the pipeline such as flushes and stalls. Control outputs are limited to a single, per-stage output which controls stage acceptance. For most stages, acceptance is defined as the latching of pipeline registers. The execution stage, however, requires more elaborate control, as some instructions do not complete in a single cycle. In these cases, each local controller can effect a pipeline stall via its interface to the pipeline controller.

The source of such partitioning originates with the construction of a basic pipeline controller. The elegance of a pyPBS interlocked pipeline specification lends itself to simple methods of extension. Compartmentalization of the pipeline controller provides a solid infrastructure from which more specialized control code can be developed and tested. This greatly eases development and debugging by limiting the scope of design errors and modifications. This strategy further allows localization of control with functional components respectively. In a larger design, this ability can greatly reduce wire delay cost associated with communication between control and data path units.

5.2 Pipeline Controller Specification

Figure 7 presents the pyPBS specification for the JackKnife pipeline controller. This controller serves as the top-level controller for the system, effecting pipeline stage acceptance, stall, and flush. The modifications necessary to construct the actual top-level controller are shown to be quite minimal in comparison with the basic pipeline presented earlier. Several additional pyPBS operators were required to specify the additional pipeline behavior. These constructs will be presented in context. For simplicity, all active low signals are shown with '_n' suffixes.

Unlike the basic pipeline, the JackKnife design supports a programming mode on reset to allow for uploading program code prior to execution. This capability is implemented in the control specification by a single wait state controlled by a programming input signal. Subsequent behavior follows from the basic pipeline, requiring only minor modification to the respective pipeline stages for processing additional stall and flush conditions. The addition of stall sources is done in a very straightforward manner through the use of the OR ('|') operator. The read stage, for instance, stalls on oc-

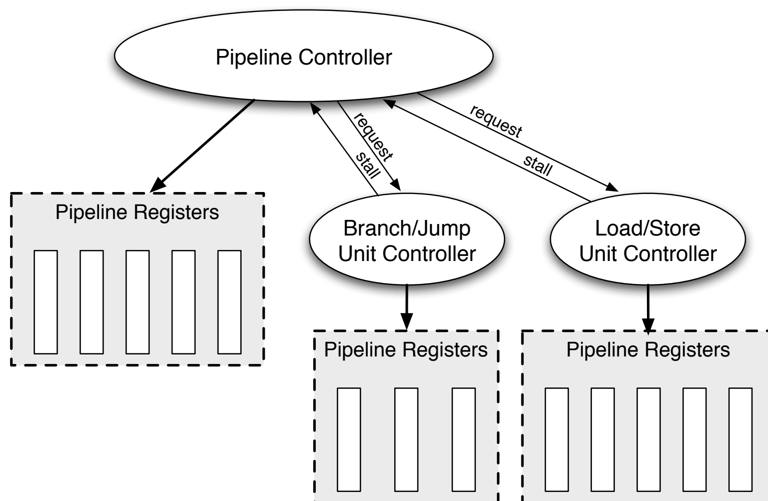

Figure 6: JackKnife control hierarchy, illustrating the division of control units and interconnects.

currence of either of the two stall input signals generated by the execute stage and data dependency logic, respectively. A complete discussion of the brackets ('[ ]') operator is beyond the scope of this paper and unnecessary for understanding the specification.

Figure 8 illustrates the resulting pipeline controller logic circuit. Each pipeline stage can be seen as a pair of registers: one signifying stage acceptance, the other stage stall. Pipeline flushing is slightly more complicated as a flush should not only clear the context from the stage's active register, but also its stall register. pyPBS provides the qualify (':') operator for such a purpose. Qualifying each pipeline stage with its respective flush signal prevents the stage from accepting or maintaining a stall context unless that signal is asserted hi (flush signals are active-low). Assertion of a flush signal prevents the read stage, for instance, from preserving an active control point regardless of other inputs. The result is that stalled or executing instructions are swept from the pipeline stage. With even a basic understanding of the pyPBS syntax, one can see that adding additional stall and flush conditions is trivial, making the specification quite extensible. Further, the synthesized pipeline controller is shown to be well suited for high-performance design by its 1.8GHz+ cycle frequency, and meager 3,000μm^2 in 0.15μm TSMC.

5.3 Branch/Jump Unit Controller Specification

The branch/jump unit is responsible for all PC-updating instructions. These instructions complete in one or more cycles depending on instruction complexity and memory latency. Partitioning the control allows the specification for this unit to be specifically targeted to its respective operations without concern for the rest of the system. This provides a compact specification that is easier to design and verify. A pair of signals facilitates communication with the pipeline controller for initiating requests and pipeline stalls. The specification for the branch/jump unit is longer than that of the pipeline controller due to the fact that it handles a greater variety of operations.

Rather than operating in a pipelined fashion, the control for the branch/jump unit allows at most one instruction to be processed at a time. In Figure 9, the top production 'bju' enforces this behavior, idling until a request is made. After servicing this request, control loops back to the idle state via the one-or-more ('+') operation. The subsequent 'cleanup' stage is executed only when control will return to the idle state – as seen by the condition for this stage. In such a case, busses must be released and switching logic reset. However, if two requests occur back-to-back, cleanup is unnecessary, as the control for the subsequent instruction will perform any such tasks.

Request handling varies based on the instruction type. In the 'op' production, control is distributed to type-specific control fragments. The conditions for each of the fragments are mutually exclusive, resulting in only a single accepting production. Each of these productions is aptly named for its respective instruction type. Half of the instructions (jump, branch immediate, skip) are trivial to specify, requiring only a single stage to perform necessary updating of the program counter. The remaining instructions, however, perform more complex services. The call and return instructions, for instance, interface with the load/store unit and therefore specify wait states which terminate upon successful completion of any necessary memory operations. The specification for these instructions is a simple string of events throttled by the load/store unit ready signal. The remaining delayed branch instruction, specified as two cycles, requires the additional cycle for branch target calculation.

5.4 Load/Store Unit Controller Specification

The load/store unit is responsible for non-instruction-fetch memory operations. This amounts to all memory instructions executed by program code. Interfaces to the processor data bus and branch/jump unit support standard load and store operations, and call and return operations, respectively. All memory interfaces use a synchronous protocol where stores are single cycle and loads assume that data is valid on the next cycle. The addition of a memory wait input allows dynamic wait states, supporting a range of memories

```
machine pipeline_controller {
        input Programming_n,
              Stall_Dependency,
              Stall_Lsu,
              Stall_Bju,
              Flush_Schedule_n,
              Flush_Fetch_n,
              Flush_Decode_n,
              Flush_Read_n;

        output Schedule, Fetch, Decode, Read, Execute, Commit;
        output Stall_Schedule, Stall_Fetch, Stall_Decode, Stall_Read, Stall_Execute;

        control ->  program , .* , cycle;
        cycle ->    schedule , fetch , decode , read , execute , commit;

        program  -> (~Programming_n*) , Programming_n;
        schedule -> (~(Stall_Fetch <Stall_Schedule>+)) <Schedule>;
        fetch    -> (Flush_Schedule_n : (~(Stall_Decode <Stall_Fetch>+))) <Fetch>;
        decode   -> (Flush_Fetch_n : (~(Stall_Read <Stall_Decode>+))) <Decode>;
        read     -> (Flush_Decode_n : (~([Stall_Execute | Stall_Dependency]<Stall_Read>+))) <Read>;
        execute  -> (Flush_Read_n : (~([Stall_Lsu | Stall_Bju]<Stall_Execute>+))) <Execute>;
        commit   -> . <Commit>;
}
```

Figure 7: pyPBS specification for JackKnife pipeline controller as created through simple extension of the basic pipeline controller via the addition of specialized recognizers.

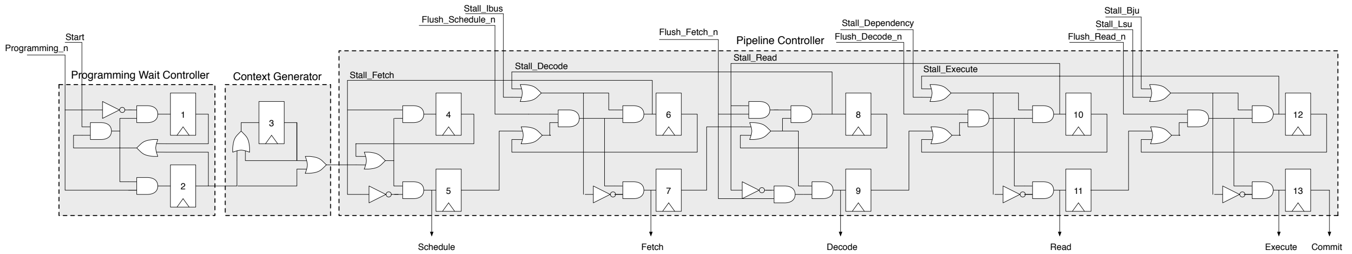

Figure 8: Generated circuit for the JackKnife pipeline controller.

that may be connected to an external memory controller.

The AVR instruction set supports a range of pre-decrement, post-increment, and displacement memory access instructions. These variations are supported in the processor pipeline, where address calculations are performed by the processor ALU. Access to instruction memory must be arbitrated to avoid bus contention between the fetch and load/store units. In this scenario, priority is given to the load/store unit to allow completion of dispatched instructions.

The specification in Figure 10 behaves similarly to that of the branch/jump unit presented above. Only a single request can be serviced at a time as enforced by DFA-like behavior in the the 'lsu' production. Unlike the pipeline controller specification, which generates an active point of control (context) on every cycle, this unit loops back to an accepting state only after termination of any active operations. Requests are subsequently distributed among four task-specific control fragments, whose mutually-exclusive conditions ensure that only a single point of control will follow. To avoid bus contention, all buses should be released upon completion of memory operations. This task is handled by 'cleanup' when the unit does not have a pending request. This is of particular importance to instruction memory where instruction fetch would deadlock.

Load and store operations are shown to be trivially specified with stores executing in a single cycle and loads operating in two or more cycles depending on memory latency. Requests from the branch/jump unit, however, require a greater number of states due to the width mismatch of instructions and memory. A 16-bit branch-requested operation requires two 8-bit memory operations – each of these operations providing support for dynamic wait states. While the same behavior could be completed by signaling two consecutive push or pop operations, this method allows tighter control handling and reduced complexity in the data path. This is of particular interest for return operations, where it is possible to pipeline pops to reduce total latency. The branch/jump unit is subsequently notified upon completion of requested loads and stores.

6. DESIGN EXTENSIBILITY

Our design methodology has shown to offer numerous advantages for the specification of control structures. We have suggested that the separation of control and data path specifications, coupled with the compact nature of the pyPBS language allows these designs to be easily extended. To support this argument, we show how several different multicycle multipliers can be swapped into the JackKnife design with minimal effort. In a traditional specification, replacing the single stage multiplier could prove a challenging task, as

```
machine bju_controller {
        input Request_n, Lsu_Ready_n, Branch_Wait_On_Status,
              Jump_Op, Branch_Op,
              Call_Op, Return_Op, Skip_Op;

        output Idle, Jump_Accept, Branch_Accept1, Branch_Accept2, Branch_Latch,
               Call_Accept1, Call_Accept2, Call_Wait,
               Return_Accept1, Return_Accept2, Return_Wait,
               Skip_Accept1, Cleanup,
               Stall_Pipeline;

        bju          -> (Request_n<Idle>* ,
                        op)+ ,
                        cleanup;
        op           -> jump | brnch | call | ret | skip;
        jump         -> [~Request_n & Jump_Op]<Jump_Accept>;
        brnch        -> brnch_delay | brnch_imm;
        brnch_imm    -> [~Request_n & Branch_Op & ~Branch_Wait_On_Status] <Branch_Accept1>;
        brnch_delay  -> [~Request_n & Branch_Op & Branch_Wait_On_Status] <Branch_Latch> ,
                        . <Branch_Accept2 | Stall_Pipeline>;
        call         -> [~Request_n & Call_Op] <Call_Accept1> ,
                        Lsu_Ready_n<Call_Wait | Stall_Pipeline>* ,
                        ~Lsu_Ready_n<Call_Accept2 | Stall_Pipeline>;
        ret          -> [~Request_n & Return_Op] <Return_Accept1> ,
                        Lsu_Ready_n<Return_Wait | Stall_Pipeline>* ,
                        ~Lsu_Ready_n<Return_Accept2 | Stall_Pipeline>;
        skip         -> [~Request_n & Skip_Op]<Skip_Accept1>;
        cleanup      -> Idle<Cleanup>;
}
```

Figure 9: pyPBS specification for the JackKnife branch/jump unit.

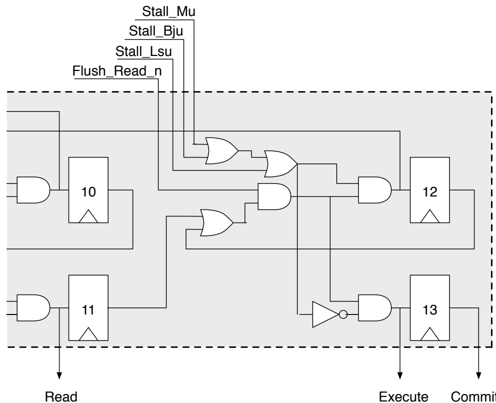

Figure 12: Generated circuit of the JackKnife pipeline controller highlighting the modifications required for supporting a multi-cycle multiplier.

it changes the timing of a number of components. In pyPBS, however, modifications are limited to a simple change to the pipeline controller and the addition of a multiplier control unit. Modifications to the pipeline controller consist of the addition of a multiplier stall input source, as highlighted in Figure 11. As expected, the resulting circuit level changes are minimal, as shown in Figure 12. The resulting pipeline controller is completely general, supporting a variety of multi-cycle multipliers without additional modification.

The multipliers, on the other hand, require specialized control units. Figure 13 presents several control specifications for multi-cycle multipliers. All of the specifications use the same request/stall control communication architecture presented earlier to interface with the pipeline controller. The first specification supports a simple 2-stage pipelined multiplier. While this controller could be trivially specified in any HDL, it is presented in pyPBS for completeness. The controller can accept a request on every cycle, producing an intermediate output signal in the same cycle and completion output signals on the final cycle. While other functional units protected against simultaneous execution of multiple operations, this is unnecessary for a pipelined multiply allowing simplification of the specification. Generation of the pipeline stall signal is shown to be easily implemented within the specification.

The second specification uses the pyPBS repetition (‘ ˆ ’) operator to extend the multiplier pipeline to three execution stages. The controller is additionally extended to provide a fourth cycle for latching input operands. The third specification describes the more complex behavior of a dynamic-length multiplier. The number of stages in such a multiplication operation is dynamically determined by operand values. The controller for this multiplier relies on a completion signal to be generated by the data path. Such a signal may also serve the function of controlling pipeline interlocking on adjacent stages or triggering other processor events. Because dynamic-length operations cannot provide static latency, this controller does not allow pipelining and therefore limits execution to a single operation at a time. A pipelined implementation could be realized by extending the basic interlocked pipeline specification to accommodate the maximum latency case.

While our presented architecture aims to avoid the complexities of out-of-order execution by stalling the pipeline for long latency instructions, the addition of multi-cycle functional units suggests that a more opportunistic approach be taken. The added complexity in specifying this type of architecture comes in the form of a more complicated commit unit, potentially requiring a reorder buffer and supporting

```
machine lsu_controller {
        input Request_n, Memory_Wait_n,
             Load_Op, Store_Op,
              Bju_Request_n, Push2_Op_n, Pop2_Op_n;

        output Idle, Load_Output_Address, Ld_Input_Data, Store_Output_Address_Data,
               Push2_Accept1, Push2_Accept2,
               Pop2_Accept1, Pop2_Accept2, Pop2_Accept3,
               Memory_Wait1, Memory_Wait2, Memory_Wait3, Cleanup,
               Stall_Pipeline;

        lsu      -> (Request_n<Idle>*,
                    (ld | st | bju))+ ,
                    cleanup;
        ld       -> [~Request_n & Load_Op]<Load_Output_Address> ,
                    (~Memory_Wait_n)<Memory_Wait1 | Stall_Pipeline>* ,
                    Memory_Wait_n<Ld_Input_Data | Stall_Pipeline>;
        st       -> [~Request_n & Store_Op]<Store_Output_Address_Data>;
        bju      -> bju_push | bju_pop;
        bju_push -> [~Bju_Request_n & ~Push2_Op_n]<Push2_Accept1> ,
                    . <Push2_Accept2 | Stall_Pipeline>;
        bju_pop  -> [~Bju_Request_n & ~Pop2_Op_n]<Pop2_Accept1> ,
                    (~Memory_Wait_n)<Memory_Wait2 | Stall_Pipeline>* ,
                    Memory_Wait_n<Pop2_Accept2 | Stall_Pipeline> ,
                    (~Memory_Wait_n)<Memory_Wait3 | Stall_Pipeline>* ,
                    Memory_Wait_n<Pop2_Accept3 | Stall_Pipeline>;
        cleanup  -> Idle<Cleanup>;
}
```

Figure 10: pyPBS specification for the JackKnife load/store unit.

logic.

Any of these multipliers can be swapped into the JackKnife design with minimal effort. In a conventional HDL, such modifications would be much more difficult, requiring significant modifications to a number of design components. Specification of even simple control in pyPBS is beneficial as it provides a compact form where behavior is well understood, extension of which is straightforward. The expressiveness of the language clearly allows control structures to take shape quickly. Further, the sparse encoding of generated machines is well suited for high performance designs. Our JackKnife design synthesizes at over 400MHz in 0.15μm TSMC standard cells – more than 20 times the maximum clock rate of commercially available AVRs.

7. RELATED WORK

pyPBS[9] belongs to a set of languages which impose synchronous timing behavior on their programs. There are several such languages, each with differing capabilities: State Charts [6] provides an efficient mechanism for state machine hierarchy specification, Esterel [2] allows for finite state machine composition in imperative language form while the data-flow oriented languages Signal [1] and Lustre [4] allow declarative specification of sequencing, and BlueSpec [7] allows specification in terms of atomic actions. The general notion of a clock enabled variable in Signal and Lustre is similar to the notion of context in pyPBS, with the exception that there is no notion of variable assignment or of variable dependence in pyPBS. Thus, the substantive complexity of dependence following and related correctness procedures are missing. pyPBS attempts to mirror the minimal complexity construction of a set of timing signals which are used to control data-flow activity. It is thus similar to pure Esterel and State-Charts in that functional activity is not modeled. In practice, correctness modeling can be prohibitively expensive for complex machines. Instead of making such behavior the default, pyPBS supports construction of static constraints or dynamic monitors [12] as required for validation and design debugging.

Previous work by Seawright [13] described a high-level synthesis system based on the hierarchical Production-Based Specification (PBS). The underlying abstraction model was extended regular automata directly translated to circuit form. It avoided representation explosion through direct construction from an abstract syntax tree (AST), avoiding the RE to NFA conversion. This approach was shown to produce favorable machine encoding when compared to existing graph-based techniques in terms of logic complexity and depth. Crews [5] showed that PBS could be used to perform sequential optimization and construct machines that exhibit synthesis complexity scaling in terms of the number of state bits rather than the number of states. His optimizations included simple heuristics on the synthesized circuit and relatively powerful RE-based simplification on the AST. Finally, Seawright [14] described a system for partitioning the AST to provide a bridge to conventional sequential synthesis techniques that was particularly useful for counter and timer applications. These techniques resulted in hardware machine implementations that were comparable or superior to those synthesized using conventional methods and which could be scaled to very large (thousands of bits) FSMs. This work was the progenitor of the Synopsys Protocol Compiler [11] and pyPBS [9]. Drawing from a general construction method, pyPBS added several powerful language constructs necessary for specifying complex control operations such as tail-recursion and non-local observation. This work greatly expands upon earlier pipeline methodologies [9] with a new interlocked design and a general methodology for implementing complex processor control.

Berry and Gonthier [2] proposed Esterel, a synchronous language for reactive programming. This language provides a means for non-deterministic machine description based on "reactive" recognition of inputs. From these independent reactions, a non-deterministic finite automata (NFA) model

```
machine pipeline_controller {
        input Programming_n,
              Stall_Dependency,
              Stall_Lsu,
              Stall_Bju,
              Stall_Mu,
              Flush_Schedule_n,
              Flush_Fetch_n,
              Flush_Decode_n,
              Flush_Read_n;

        output Schedule, Fetch, Decode, Read, Execute, Commit;
        output Stall_Schedule, Stall_Fetch, Stall_Decode, Stall_Read, Stall_Execute;

        control ->  program , .* , cycle;
        cycle ->    schedule , fetch , decode , read , execute , commit;

        program  -> (~Programming_n*) , Programming_n;
        schedule -> (~(Stall_Fetch <Stall_Schedule>+)) <Schedule>;
        fetch    -> (Flush_Schedule_n : (~(Stall_Decode <Stall_Fetch>+))) <Fetch>;
        decode   -> (Flush_Fetch_n : (~(Stall_Read <Stall_Decode>+))) <Decode>;
        read     -> (Flush_Decode_n : (~([Stall_Execute | Stall_Dependency]<Stall_Read>+))) <Read>;
        execute  -> (Flush_Read_n : (~([Stall_Lsu | Stall_Bju | Stall_Mu]<Stall_Execute>+))) <Execute>;
        commit   -> . <Commit>;
}
```

Figure 11: pyPBS specification for the JackKnife pipeline controller including modifications required for supporting a variety of multi-cycle multipliers.

can be recognized and translated into a deterministic state graph. Modern Esterel compilers avoid the potential explosion of the translation process and can produce high quality sequential hardware designs. In general, however, the bulk of Esterel application has been in reactive software systems.

Both PBS and Esterel provide a means for "safe" machine composition, which is required to create large designs. The notion of "safe" describes the property that a target automaton's protocol is not changed by inclusion in a larger construction. In Esterel, this is managed by enforcing a single point of control model within a control structure. This is familiar to software designers and makes for efficient translation of Esterel code to machine instructions. In contrast, PBS use of regular automata is "safe" in the opposite sense: each submachine is assumed to be as general as any possible invocation sequence can make it. In other words, any PBS production is designed under the assumption that a new invocation may occur each cycle. No attempt is made to preserve the potentially lost points of control. This second view is a close match to the actual behavior of logic circuits. Thus, while Esterel can be compiled into efficient hardware, many varieties of pipeline behavior are difficult to specify and synthesize efficiently. Esterel, on the other hand excels in error handling and modal changes, which need relatively complex primitives in pyPBS.

Hoe and Arvind [7] proposed an operation-centric hardware abstraction model useful for describing systems that exhibit a high degree of concurrency. In this model, operations are specified as atomic actions, permitting each operation to be formulated as if the rest of the system were frozen. A recent extension of BlueSpec [8] proposes a forcing operator enabling sequential constraint of the output design. Despite similarity of application, pyPBS and Bluespec serve substantially different needs. pyPBS provides a practical specification and synthesis strategy for extremely complex sequential protocols and related systems. Much of the power of BlueSpec is lost in such a scenario. On the other hand, Bluespec allows practical reasoning about complex operation dependency which is not supported in pyPBS. In the middle ground, both languages have been used to describe microcontrollers, with Bluespec providing succinct emulation of Itanium scale designs. In contrast, pyPBS is used in this paper to concisely design a multi-stream processor with precisely defined interfaces, both to external devices and to internal sequential peripherals.

The IBM/Accellera Formal Specification language PSL/-SUGAR [3] has become the basis for formal property specification in System Verilog. This language contains extended regular expression assertions as well as CTL and LTL model checking capabilities. Although PSL is not intended for design implementation, the similarities show that formal property checking can be done in this format. There was a property checking extension of PBS in its Protocol Compiler instantiation as well.

8. CONCLUSIONS

Our design methodology aims to ease the burden of design and development of pipelined or highly sequential systems by allowing construction of complex control through incremental extensions to a proven interlocked pipeline specification. Support for a number of stall and flush behaviors can be added through addition of specialized recognizers with minimal effort. By encouraging specification of simple, compartmentalized control, pyPBS provides insight into viable partitioning strategies and allows simple integration of distributed control. Implementation of a fully synthesizable, multi-threaded AVR has shown real specifications to be compact, expressive, and extensible. Further, we have shown that traditionally difficult modifications, such as swapping multipliers, can be easily accomplished. We believe that our methodology provides a framework for reduced cost implementations allowing for application of more aggressive architectural techniques.

```
machine mu_controller_piped_two_cycle {
        input Request_n;

        output Stall, Latch_Intermediate, Latch_Result;

        mu   -> .*, ~Request_n <Latch_Intermediate> , . <Stall | Latch_Result >;
}


machine mu_controller_piped_three_cycle {
        input Request_n;

        output Stall, Latch_Operands, Latch_Intermediate, Latch_Result;

        mu   -> .*, ~Request_n <Latch_Operands> ,
                     . ^ 2 <Stall | Latch_Intermediate> ,
                     . <Stall | Latch_Result >;
}


machine mu_controller_variable_length {
        input Request_n, Mult_Complete;

        output Stall, Latch_Operands, Latch_Intermediate, Latch_Result;

        mu   -> (Request_n <Idle> + , ~Request_n <Latch_Operands > ,
                      (~Mult_Complete) <Stall | Latch_Intermediate> + ,
                      Mult_Complete <Stall | Latch_Result >)+;
}
```

Figure 13: pyPBS specifications for a 2-cycle pipelined multiplier, a 4-cycle pipelined multiplier, and a variable length multiplier.

9. REFERENCES

[1] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[2] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[3] D. F. C. Eisner. Sugar 2.0 proposal presented to the accellera formal verificaition technical committee.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, Jan. 1987.

[5] A. Crews and F. Brewer. Controller optimization for protocol intensive applications. In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 140–145, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[7] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 511–519, Piscataway, NJ, USA, 2000. IEEE Press.

[8] J. C. Hoe and G. Nordin. Synchronous extensions to operation-centric hardware description languages. In *MemoCODE 04*, June 2004.

[9] G. Hoover and F. Brewer. Pypbs design and methodologies. In *Third International Conference on Formal Methods and Models for Codesign*, pages 55–64, 2005.

[10] G. Hoover and F. Brewer and T. Sherwood. A Case Study of Multi-Threading in the Embedded Space. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2006.

[11] W. Meyer, A. Seawright, and F. Tada. Design and synthesis of array structured telecommunication processing applications. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 486–491, New York, NY, USA, 1997. ACM Press.

[12] M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of ip interface monitors. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 129–134, New York, NY, USA, 2002. ACM Press.

[13] A. Seawright and F. Brewer. Clairvoyant: A system for production-based specifications. In *IEEE Transactions on VLSI Systems*, volume 2, pages 172–185, June 1994.

[14] A. Seawright, J. Buck, U. Holtmann, W. Meyer, B. Pangrle, and R. Verbrugghe. A system for compiling and debugging structured data processing controllers. In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 86–91, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[15] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the tera mta. In *1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–8, 1998.