Symbolic NFA Scheduling of a RISC Microprocessor

Forrest Brewer, Steve Haynal

Abstract—We describe a set of techniques for representing the high-level behavior of a digital subsystem as a collection of non-deterministic finite automata, NFA. Desired behavioral dynamics such as functional dependencies, sequential timing and sequencing, and control state are similarly modeled. Using techniques similar to that used in Formal Model Checking, we implicitly explore the possible execution sequences of the system, obeying all imposed constraints. This provides a very general, systematic mechanism for performing high-level synthesis of cyclic, control dominated behaviors constrained by arbitrary sequential constraints. In this paper, we show that these techniques are scalable to practical problem sizes and complexities by constructing a high-level model of a (MIPS IV) RISC Microprocessor and then performing exact scheduling and related design tradeoffs on this model. The model is constructed at the level of register transactions to address the majority of contention and arbitration issues of architectural interest.

I. INTRODUCTION

A. Scope

In this paper, we present an approach to the high level scheduling of a MIPS-based microprocessor at a level of abstraction which allows direct tradeoffs for contention for resources and pipeline stage sequencing of internal computation, memory and register file access, bus interfacing and speculative behavior. The model is appropriate for designer exploration of architectural alternatives while providing mechanisms for accommodation of realistic constraints such as interface protocols, memory access sequencing, and several forms of resource contention. Processor performance evaluation is made possible by reference to benchmark program trace information (e.g. Simple Scalar[2], Media Bench[10]) which is used to weight the timing of execution sequences and is also used in the optimization process itself. In contrast to most previous work, we do not impose a pre-specified processor model during the synthesis. Instead, the possible execution sequences arise solely from resource constraints, implicit dependency analysis (including many forms of speculation) and sequential constraints, typically on interfaces.

Scheduling processors within a high level abstraction is problematic as there are several issues in processor architecture making synthesis difficult. First, processors are subject to a wide variety of sequential constraints such as bus interface sequencing, activity on internal reconfigurable pipelines, and pipelined access to register files. Second, the level of optimization required for processor design is very high. This is due to the economy of selling large numbers of hardware parts as a commodity. Third, the control behavior of processors is very complex given data and control hazards, pending and interrupted instructions, and architecturally mandated control behavior.

In order to proceed despite these difficulties, we choose to solve a subset of the general high level synthesis problem by restricting both the problem specification and its solutions to those behaviors that can be modeled by Non-deterministic Finite Automata (NFA). This decision is motivated by the finite nature of practical hardware, the similarity of NFA transition models to hardware execution and the large supply of efficient tools and techniques developed for formal verification problems. In a manner akin to verification, we model both the design hardware resources and the design constraints via specially constructed NFA models. Unlike verification, we model behavioral constraints and dependencies as well as resource contention and sequencing. Further, we are free to choose the structure of the automata to simplify the representation issues.

The automata are used to construct a composite model (CMA) which implicitly includes every possible execution sequence. From this model the latency and resource requirements for any valid schedule can be determined. However, a complete (ensemble) schedule for a system with control and loops must obey several additional constraints: 1. All schedules must be repeatable, i.e. the final state for some repeating sequence must be a valid starting state for another. 2. At any time in the execution, all possible values of control variables must be accommodated by some execution sequence. 3. At any time in the execution, there must be a unique scheduling step in the current cycle. i.e. the schedule must be deterministic. A disadvantage of this approach is that such a model is potentially intractable. However, we will demonstrate that by judicious choice of models and encodings, high level synthesis problems of unmatched scale and complexity can be scheduled exactly on relatively modest computers.

Due to the brevity of this paper, we cannot adequately reference the large variety of previous related work, so provide only references for the most comparable work. In the second section, we provide an overview of symbolic scheduling motivated by a simple example. The MIPS processor model is presented in section III and results based on the exploration of this model are presented in section IV.

B. Related Work

Work in processor and system synthesis includes synthesis on FPGA arrays[1], Term Re-writing[8], Symbolic Simulation methodology[5], the CFMS models of POLIS/ECL[9], and pipeline synthesis[3]. A common thread in these systems is the
use of heuristic scheduling. This decision is made for reasons of design scale as well as technical problems in exact scheduling algorithms. None of these techniques support sequential interfacing or protocol accommodation. Instead, they rely on abstractions which potentially affect the quality of results.

Relevant work in symbolic scheduling algorithms includes that of Radivojevic[11], Cohello[4], and Yang[13]. In symbolic scheduling, formal logic based models are used to implicitly describe the scheduling problem. While they are comprehensive, they cannot be used for practical sized designs due to representation growth. In our work, there are still representation complexity issues, but by use of clever encodings and partitioning, we can address problems of practical significance, as demonstrated by this paper.

II. SYMBOLIC SCHEDULING OVERVIEW

We model the sequential production of operands in a digital system. Each operand is created by some resource when the preconditions of its dependent operands and the resources’ sequential behavior are met. For each operand, we construct an NFA modeling automata (MA) to represent this process. The MA is non-deterministic in the sense that once its requirements are met, it might or might not transition to a state in which the resources are used and production of the related operand is initiated. Eventually, the operand is produced and becomes known. Before that state is reached, the operand is unknown and does not satisfy dependency requirements of other MA. During the execution of a resource, contention is modeled by one or more busy signals which are used to construct constraints on the initiation or sequencing of other MA. By this means, a functional model of a digital system is constructed with MA states representing the existence of operands and several varieties of boolean constraints modeling the dependencies, resource utilization, and control values. The sequencing characteristics of the MA model the sequential timing of the of the modeled resource subsystem. The composite model for a system is called a CMA which is constructed by making a cartesian product automata of the MA followed by application of the constraints. State sequences of the CMA describe valid execution traces of the model, where a trace is some particular control path of the system.

Control branching and looping behavior are modeled by introducing control labels which mark the execution of a MA as required (if the control is true) or don’t care (if the control is false). Control values are themselves operands and their production is mediated by an appropriate MA. Thus control values are unknown prior to the associated MA execution, and are known thereafter. Control dependencies are only made to ‘join’ or select nodes in the CDFG allowing for arbitrary speculation past control ‘forks’ or branches. Looping behavior is modeled by a mechanism similar to software pipelining. Each MA is constructed as a cyclic automaton which executes in one of two senses. Figure 1 shows a single-cycle MA in which both senses are encoded as a single bit. Operand ‘a’ idles unknown until it transitions to its known state. A subsequent iteration in the ~ sense interprets this state as the unknown state for the new production of ‘a~’. Looping behavior is then modeled by multiple copies of the CMA, one for each iterate. Senses allow the MA to disambiguating which iterate it is executing and each iterate designates operands from a separate iteration of a loop. The number of iterates places a limit on the number of simultaneous instances of operands from differing loop iterations which are allowed in the model. For the processor model in particular, an iterate models the complete execution of an instruction. The number of iterates indicates how many simultaneous instructions can be present in the processor at any time. Please note that this is not necessarily the pipeline depth of the processor. In the following, we present a very abbreviated overview of symbolic techniques and an illustrative example. For more detailed and general accounts the interested reader is directed to [6], [7].

Although the CMA so constructed has all possible behaviors, it also has many that are not possible, causal, or desired. Constraints on the automata come in 2 general types: state and transition constraints which can be implemented by pruning the allowed transitions of CMA, and path based constraints which prescribe relations between alternative paths (representing scheduling control traces or alternative sequencing). The second class of constraints typically require constructed CMA state sets or supersets and proceed in an iterative fashion on the sets of states constructed for each time step. Transition pruning constraints include:

A. Dependency

This constraint prunes those transitions of CMA from states in which the operand has yet to be produced to those states in which it must have been used. Note that it is possible for several alternative MA to construct a given operand, and that dependencies are control dependent.

B. Concurrency

Concurrency constraints prune state transitions which require more instances of busy signals than are allowed by the resources. They are often implemented by k-choose-n logic functions which have ROBDD size of $O(k \times n)$[6]. Concurrency provides a generalized means for controlling the allowed use of any resource such as registers, busses, function units, subsystems or interfaces.

C. Capacity

This constraint prevents a local MA from changing it’s sense until all dependent MA have consumed the information in the current sense. This enforces that only a single operand of a given sense can be known in CMA at any time. Multiple instances of an operand from differing iterations of a loop are modeled by different iterates.
Algorithm 1 Simple Example

```c
    c = 0
    while (1){
        a = IOread();
        if ((100-c) > 0)
            b = a + c;
        else
            b = a;
        c = b * b;
        IOwrite(c);)
```

D. Control

Control constraints partition behavior into portions that must be or need not be executed in reference to decisions during the execution. Dependency and capacity constraints depend on the value and known state of a control operand.

The pseudo-code in Algorithm 1 is modeled as a set of simple single-cycle automata each representing the production of one of the operands a, b, c, (ftrue, ffalse), and e = IOWrite. Figure 2 shows an explicit fragment of this CMA. Highlighted characters indicate sustainable cyclic solutions for both control resolutions. Labels on the arcs describe resource usage in that transition. For this MA, activity occurs on the transition from unknown to known. For the transition from \( \{a,b,c\} e f \) to \( \{a,b,c\} e f \), operand a has been computed for one iteration sense and c has been computed for the other iteration sense. Alternative paths occur when the control value f has been resolved to true or false values.

The inner cyclic path, passing through four states, represents continuous repetition of the algorithm given false control resolution in each iteration. For this path, throughput is 2 time-steps while latency is 4 time-steps. An iteration produces a in cycle 0, f in cycle 1, b and c in cycle 2, and e at time-step 3. If during some iteration, control resolves true, then the state which computes b on the outer cyclic path is visited. For the true path, throughput is 3 time-steps while latency is 5.

Figure 2 shows only the efficient closed path fragment of this CMA. Techniques for finding such paths in the CMA is the objective of the model exploration constraints described below. Generally, a causal ensemble of compatible looping paths covering all control resolutions must be found. To this end, the paths must also obey these additional path-based constraints: repeatability and validation:

E. Repeatable Minimum Latency Exploration

1) Loop Cutting: Loop cutting determines a set of starting states guaranteed to include states on every shortest repeating path. This avoids the problem that execution from an arbitrary start set might require a very long preamble before settling to an optimal closed path. We choose an operand common to many control paths and select every state of CMA with a transition producing that operand in the non~ sense. This set is called LCS~.

2) Execution Sequence Set: An execution sequence set is generated by symbolic exploration of CMA starting from LCS~ and concluding with LCS~. (In this example, this set is just the bitwise complement of the set LCS). Note that a path from LCS~ to LCS~ implies paths from LCS~ back to LCS~. For the example CMA in figure 2, starting from state \( \{a,b,c\} e f \) the execution sequence set is: \( \{(a,b,c,e,e,f)\} \) \( \{(a,b,c,e,f)\} \) \( \{(a,b,c,e,f)\} \). At time-step 2 (indicated by superscript), a state in LCS~ is reached.

3) Repeatable Paths: Not all of the paths in the execution sequence set are repeatable. For example, time-step 2 above contains two states. State \( \{a,b,c\} e f \) has no dual in step 0. Hence, once this state is reached, there is no path back in 2 time-steps. Practically, the set of states at time step n (TS^0) is restricted to the set of states reached at time-step n whose duals are contained in LCS~. As long as TS^0 does not equal TS^0, a fixed-point pruning is done. In the fixed-point, TS^0 is forced to equal the dual of TS^0 by a series of pre-image computations from TS^0. If the fixed point fails, the latency is incremented and exploration is restarted.

F. Validation

There are two related correctness issues which must be addressed for alternative control paths. First, all possible execution sequences must be covered for any possible set of future control values and from any state in the schedule. i.e. From any state, all future control values must have accommodating states in the schedule. Second, we must ensure that alternative control paths are indistinguishable before the control value becomes known. [11], [6] This constraint is required between every set of paths distinguished by control variables. To solve this problem would seem to require that we somehow identify all possible ensembles, which is impractical, even implicitly. Instead, we remove all transitions belonging to paths which cannot form a complete ensemble with the available paths. This weaker condition guarantees that every remaining path belongs to some complete ensemble. Validation performed this way is fragile in the sense that if we further prune the solution set, we might eliminate some or all ensembles. i.e. After further pruning, the set must be revalidated.

To efficiently implement validation, a helper state is added to each iteration sense of a control producing MA as shown in figure 3. This helper state, labeled resolve, forces evaluation of the related control value on transitions from the state. This extra state is removed from the execution model by splitting the transition relation into two sequentially active parts— a relation to mediate control activities and another representing data-flow.
dependencies and timing. From the modeling point of view, there is no imposed delay on control activity beyond what is specifically added to model design constraints.

Informally, Validation is a double fixed point (over each control variable, and each time state) which identifies all MA producing a \textit{resolve} in the next time step. For each control variable, all next state variables are universally abstracted ensuring that the paths exiting the resolve state cover all possible control values. This step is then repeated for each time-step, until a fixed point is reached. The process is a fixed point iteration since some step might remove a transition on a path which satisfied the constraint for a different time step. Validation concludes when no states are removed for any time-step.

In the example, validation guarantees that the states \((a,b,c,e,f)\) and \((a,b,c,e,f)\) in figure 2 each have two exit arcs, one for each possible control resolution, as the control information, \(f\), has just been produced. An example repeatable, complete and valid execution sequence set is: \(\{(a,b,c,e,f)\}^0, \{(a,b,c,e,f)\}^1, \{(a,b,c,e,f)_{\text{true}},(a,b,c,e,f)\}^2, \{(a,b,c,e,f)\}^3\).

G. Optimization

At this point, we have (implicitly) constructed all complete sets of minimal latency, repeatable and validated paths for the behavior. Given control dependence, we have the problem of selecting particular solutions which match the performance goals of the designer. For example, if a particular sequence of control values is far more likely in system usage, the performance of the related trace should receive priority. In general, it does not make sense to optimize the worst case path as this may well hamper the performance of more likely paths and produce an inferior (on average) design. To address this issue, a prioritization scheme has been implemented in which high priority path sets are optimized followed by optimization of lower priority paths (subject to the earlier optimizations). This can be done for arbitrary levels of priority. After each prioritization step, validation is rerun.

III. PROCESSOR AUTOMATA MODEL

A RISC processor model, implementing the MIPS IV ISA was constructed. This well-known processor has easily obtainable profiles which are used in both prioritization and evaluation of the resulting schedules. This is crucial since no processor can guarantee high throughput for every control path given contention for resources. The floating point and virtual memory subsystems are not modeled. All other instructions, including all control and all integer instructions as well as internal sequential hazards, and cache access delays.

A. Instruction Task

We describe the processor behavior as a two level composition. At the lower level of abstraction, \textit{iterates} represent execution of one entire instruction and are called instruction tasks. An instruction task sequences through a subset of behaviors depending on the decoded instruction and processor state. It represents the entire functional flow of an instruction from fetch to commit. At the higher level of abstraction, sets of iterates representing instruction tasks are modeled and sequenced. This mechanism simplifies the functional description since each iterate is identical to all others while mechanisms common to all iterates requiring global synchronization or communication are captured in the higher level. A second advantage is that the number of iterates (hence the number of modeled instructions in flight) can be easily changed and thus provides a tradeoff mechanism. In the results, we enforce sequential instruction execution as well as sequential commit.

Each instruction task computes the pre-incremented next program counter value, \(nppc\), as well as pre-fetching the next instruction, \(ninst\), at the new program counter \(npc\) address. The computation of \(nppc\) and \(ninst\) are heavily control-dependent on the currently decoded instruction and on the processor state. Finally, to avoid dynamic data hazards between values that have yet to be updated and future instructions that will read the same values, a \textit{bypass} operand is provided. Bypass allows a single operand to be forwarded to the next instruction as well as being written to the register file. Bypass can also cause processor stall behavior in cases of multiple hazards. Instruction task contain external memory and register file access tasks. These tasks are sequentially constrained, so it is convenient to represent these tasks as dedicated NFA sequential models.

B. Processor Composition

Several instruction task instances are composed to represent the behavior of simultaneously executing instructions. The example in figure 4 composes three instances to represent behavior of three in-flight instructions. Resource concurrency constraints as well as other sequential constraints and synchronization mechanisms such as bypass are applied globally in the processor composition. The resulting CMA describes execution sequences for three “in-flight” instructions. If higher performance is desired, a composition consisting of \(n\) instruction instances may be built with the trade-off of greater storage, resource, and control complexity. This technique can be generalized to permit independent control of the mechanisms for instruction fetch, execution and commitment to allow modeling of various out-of-order execution strategies.
C. Modeling Instructions

Each instruction task is composed of smaller low-level tasks. Low-level tasks include register file access, memory access, integer computations, instruction decoding and other activities. MA specify the target sequential behaviors of each of these low-level tasks. Instructions are grouped into sequentially distinct instruction classes. All instructions in a particular class require the same flow of low-level tasks with possibly different ALU or control operations. There are three classes of instructions: Integer Arithmetic, Load/Store and Control. Each class is further broken down into subclasses which describe variations based on locality of operands, data type and access mode:

1) Arithmetic and Load/Store: Arithmetic and load/store instructions comprise simple sequential behaviors distinguished by operand fetch sequencing. Exceptions are the mult and div instructions which use a special resource task representing a practical scalar multiply/divide pipeline. Other exceptions are the memory access tasks, used in load/store instructions which model memory subsystem access protocols and delays. Memory access tasks model cache hit and miss behavior. Non-determinism is used to select a short (hit) delay versus a longer (miss) delay case. Both cases are accommodated in the schedule.

2) Control instructions: Control instructions update the program counter or processor state. Program counter update and instruction pre-fetch are often bottlenecks in processor architecture due to limited fetch bandwidth. In this model, fork-type (branching) control behaviors are automatically speculated, however, join-type (operand selection) behaviors are not. To allow speculative execution of instructions, multiple instruction pre-fetches are required.

Figure 5 shows a CDFG representing instruction fetch. The left hand side is a functional flow determining the next pre-incremented pc. There are 3 cases: branch-taken, jump, or default. The right hand side of figure 5 shows the implemented CDFG supporting both speculative and non-speculative pre-fetches. This version allows the speculated pre-fetch address to be computed concurrently with verification that the current instruction decodes to the default case. The next pc, nppc, is computed speculatively when possible.

3) Data Hazards: Data hazards are supported using bypass operands. Single operand bypass is performed by generating a hazard control value during instruction decode which indicates a data hazard. In an instruction task, an operand is selected by the hazard value from either the register file or the previous instruction's write-back operand. It is still necessary to handle multiple data hazards as well. In such a case, the dependent instruction task stalls until the previous instruction task has finished all write-backs. This case is modeled with the stall control operand. If stall is true, the pre-fetched instruction has too many data hazards and is not made available to the next instruction task. The next instruction task is then forced to stall. Stall is also used to stall instructions for non-bypassed data hazards, such as hazards on architecture registers hi and low.

Both stall and hazard operands are represented by control values. Although we cannot determine these values in advance, we model when the values will become known. Modeling this behavior as control dependences allows instruction tracing data to be used to guide schedule control path prioritization including hazard and stall conditions.

IV. MODELING RESULTS

Each instruction task contains 25 MA and 37 control paths. When three instruction tasks form a processor composition, a cyclic scheduling problem with 75 tasks and 50,000+ control paths is created. This model is scheduled without imposition of other constraints to determine the fundamental limits of available performance—i.e. the only limits on execution are the fetch speculation model, the number instruction tasks and the available resources.

Based on MediaBench 10 statistics, two control prioritization sets were devised to direct the optimization of 3 iterate schedules (4 iterates gave identical CPI performance). An instruction mix consists of sets of three instructions where at least one of the instructions belongs to a prioritized class. Mix 1 favors register-to-register, register-to-immediate and branch taken instructions, followed by load single word and branch not taken. Mix set 2 favors the load single word and branch taken instructions over register-to-register, and register-to-immediate types.

Resource configurations were moderate, tight and none. The moderate configuration assumes two memory ports for all
memory read/write and instruction fetch tasks. Furthermore, at most 2 register file reads and 1 write are allowed concurrently. An integer unit is used for instruction computations and another for program counter updates. A single instruction decode unit and single access to architecture hi/lo registers are allowed. The tight configuration allows one memory port, with at most 2 reads or 1 read/1 write, and a single integer unit, shared for all computations. Two memory constraint scenarios, ‘A’ and ‘B’, are modeled. Scenario A assumes all sub-tasks complete in a single time-step except for two-word memory read/write and complex integer computations. Scenario B assumes that all memory reads require two time-steps.

Table I shows cycles per instruction, CPI, for mix 1. The moderate resource configuration achieves the same performance as no resource constraints. The ‘best’ and ‘worst’ row indicate CPI for the best and worst control paths respectively. For entries with 2 numbers, the lower number is achieved if the prior instructions come from the same or higher priority mix. Of particular interest is the ‘r1,r1,r1’ mix with scenario A, and tight resources. Because of restricted register file access, this mix can only achieve single cycle throughput if hazards exist i.e. if the value is bypassed to the next instruction instance. Since the scheduling is comprehensive, an optimal sequence for this control path which uses bypass for speed-up is found. Table II presents expected CPI data for mix set 2. Performance differences are shown in bold. Since the mix ‘lds,lds,bt’ is at a higher priority in mix 2, a solution with CPI of 4.00 is found. This solution impacts other control paths of lower priority. The ‘lds,lds,bnt’ and ‘r1,r1,bnt’ mixes are slightly worse. However, the worst case control paths benefit from this choice with CPI improving from 4.67 to 4.00.

Required CPU times range from 2 minutes to 30 minutes. Results in tables I and II were produced on a 866 MHz PIII Xeon processor with 2 GB of memory, running under Linux. The package is implemented in Python and uses the CUDD[12] ROBDD package via PYCUDD. Peak exploration state representations and transition relation sizes are shown for several configurations in table III.

V. CONCLUSIONS

In this work, we have described a modeling paradigm based on NFA techniques which allows systematic exploration of high-level design alternatives via symbolic scheduling. These techniques greatly generalize the types of problems that can be modeled in a high level abstraction while constructing exact solutions for these very highly constrained problems. Using a simple ISA meta-model for the MIPS-IV processor, we constructed optimal operation schedules for this large example, even in the face of complex sequential and control constraints. In particular, we exactly solved the behavioral scheduling problem for a system with hundreds of thousands of concurrent control paths using relatively modest computation resources. To our knowledge, systematic solution of high-level scheduling problems of this complexity has not been achieved previously. The method allowed the use of practical 2-1 register files, single hazard forwarding, and instruction caches while achieving CPI values similar to those of comparable to reference (human designed) implementations. We would like to acknowledge the support of the National Science Foundation, the University of California MICRO Program, and Intel Corporation.

VI. REFERENCES

REFERENCES


