Execution Models for Processors and Instructions

Florian Brandner
COMPSYS, LIP, ENS de Lyon
UMR 5668 CNRS – ENS de Lyon – UCB Lyon – INRIA
Email: florian.brandner@ens-lyon.fr

Viktor Pavlu and Andreas Krall
Institute of Computer Languages
Vienna University of Technology
Email: {vpavlu,andi}@complang.tuwien.ac.at

***Abstract*—Modeling the execution of a processor and its instructions is a challenging problem, in particular in the presence of long pipelines, parallelism, and out-of-order execution. A naive approach based on finite state automata inevitably leads to an explosion in the number of states and is thus only applicable to simple minimalistic processors.**

During their execution, instructions may only proceed *forward* through the processor's datapath towards the end of the pipeline. The state of later pipeline stages is thus independent of potential hazards in preceding stages. This also applies for data hazards, i. e., we may observe data by-passing from a later stage to an earlier one, but not in the other direction.

Based on this observation, we explore the use of a series of *parallel finite automata* to model the execution states of the processor's resources individually. The automaton model captures state updates of the individual resources along with the movement of instructions through the pipeline. A highly-flexible synchronization scheme built into the automata enables an elegant modeling of parallel computations, pipelining, and even out-of-order execution. An interesting property of our approach is the ability to model a subset of a given processor using a sub-automaton of the full execution model.

I. Introduction

Formal models of the dynamic behavior of processors and instructions during their execution find applications in a wide range of development, design, validation, and verification tasks. For example, cycle-accurate instruction set simulators rely on an execution model in order to efficiently emulate the processor's behavior during the execution of a program. During the simulation pipeline effects, resource-, control-, and data-hazards need to be detected and faithfully modeled in order to guarantee correctness. The validation and verification of a processor poses similar requirements. Here the processor designer seeks for a proof that the behavior of the processor actually reflects the properties stated by a formal specification. In addition, processor models can help to build software development and analysis tools, such as *worst-case execution time* (WCET) analysis tools, code profiling tools, code certification tools, and compilers.

Naive approaches based on *finite state automata* (FSA) inevitably lead to a rapid growth in the number of states. In particularly when multiple instructions are executed in parallel, as is the case for pipelined, superscalar, or explicitly parallel processors such as VLIW machines. Stalls due to various kinds of data hazards, long memory latencies, shared resources, etc. further increase the number of states, rendering this approach impractical for many application scenarios. For example, the Pentium 4 allows up to 128 instructions to be in-flight simultaneously [1].

We thus propose a partitioning of the resources of the processor's datapath, where each resource is assigned a dedicated sub-automaton that is (largely) independent of the automata of other resources. This allows the reduction of the overall number of automaton states, but requires a coordinated procedure in order to model state transitions among the individual automata. However, for most pipelined processors this partitioning can be derived readily from the pipeline organization based on the following observation: During execution the state of a resource depends only on the current instruction it is executing, its successors in the datapath where the current instruction will proceed after finishing its computations at the resource, as well as the next instruction that the resource will receive from one of its predecessors in the datapath. A similar interaction pattern can be observed for data by-passing, where register values are forwarded from instructions that are executed late in the pipeline to instructions in early pipeline stages. It is thus possible to model the state of the processor by a series of *parallel finite automata* (PFA) [2], which are synchronized in order to perform state updates according to the datapath and the processor's pipeline structure.

Furthermore, our approach allows to derive execution models of individual instructions or groups of instructions using sub-automata that include only those resources that are required for the execution of the considered instructions. This is particularly interesting for analysis and verification tasks where the reduced size of the sub-automata is expected to be beneficial.

The rest of this paper is organized as follows. First, we will present previous work on execution models in Section II, followed by an introduction to the basic concepts of parallel finite automata in Section III. Section IV introduces our execution model for processors based on parallel finite automata. Next, we present how execution models for a subset of the processor's instruction set can be derived down to the level of individual instructions. Finally, we will conclude by shortly discussing the merits our new approach in Section VI.

II. Related Work

Execution models for processors are often developed in combination with processor description languages (PDL) [3]. Lisa [4] models pipeline effects using extended reservation tables called L-charts that contain operations and the activation

978-1-4244-8973-2/10/$26.00 ©2010 IEEE

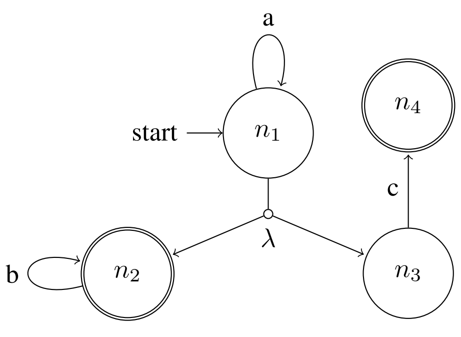

Fig. 1. A parallel finite automaton accepting $a*\ (b* \parallel c)$ - see Example 1.

relations between them to support dynamic scheduling for simple in-order pipelined processors. MADL uses extended finite state machines with tokens [5]. The token manager, however, is implemented in C/C++, so the formal model remains incomplete. The approach presented in this paper was also designed in conjunction with a PDL [6], [7]. The language relies on hypergraphs [8] in order to specify the hardware structure of a processor. Due to the close relationship between hypergraphs and PFAs it is straightforward to derive an execution model from a processor specification.

Furthermore, processor models are used in software and hardware verification tools, analysis tools, and instruction set simulators. Reshadi and Dutt use *Reduced Colored Petri Nets* to formally model the pipeline of processors. They are able to synthesize efficient instruction set simulators [9] for a wide range of processor architectures. Thesing [10] uses finite state machines in order to capture the execution state of a processor for worst-case execution time (WCET) analysis. *Synchronous Transition Systems* are used by Damm and Pnueli [11] to verify machines with out-of-order execution by showing that they reach the same final state as purely sequential machines.

III. Parallel Finite automata

Parallel finite automata (PFA) are related to traditional finite state automata, but allow multiple *nodes* of the automaton to be active simultaneously. Stotts and Pugh showed that PFAs are equivalent to *finite state automaton* (FSA) [2], i.e., it is possible to construct a regular FSA from any given PFA. However, this transformation may lead to an exponential growth in the number of states. PFAs thus allow a more compact representation of certain classes of regular languages.

Definition. A **parallel finite automaton** is defined by a tuple $M = (N, Q, \Sigma, \mu, q_0, F)$, where N is a set of nodes, $Q \subseteq \mathcal{P}(N)$ is a finite set of states, Σ a finite input alphabet, $\mu : \mathcal{P}(N) \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(N)$ is a transition function, $q_0 \in Q$ a start state, and $F \subseteq N$ a set of final nodes.

Very similar to traditional automata, the execution of a PFA is described by repeated transitions from one state to another. The difference is that multiple nodes can be active on each state before and after every transition. Consequently, the transition function μ provides an extended semantics covering multiple nodes of the automaton. Transitions with multiple target nodes are termed *parallel* transitions, while those having multiple source nodes are termed *synchronization* transitions. A transition can be applied, or is *enabled*, if all its source nodes are active in the current state and the current input symbol matches the transition's label. The new state after the execution of a transition is derived by first deactivating all source nodes and then activating all its target nodes. At the same time the current input symbol is consumed, unless the label represents the special symbol 'λ'. In that case the current input symbol is not consumed, nevertheless the transition is still considered enabled. The automaton accepts its input when all the input symbols have been consumed and a final state has been reached, i.e., when all final nodes in F are active. A formal definition of PFAs and their execution is presented in [2].

A PFA can be represented as a labeled directed hypergraph [8] $\mathcal{H} = (V, E)$, consisting of a set of vertices V and hyperedges $e \in E \subseteq \mathcal{P}(V) \times \mathcal{P}(V) \times (\Sigma \cup \{\lambda\})$. The automaton's nodes are represented by vertices of the graph, i. e., $V = N$, and the transitions in μ correspond to labeled hyperedges.

The language accepted by a PFA can be represented using an extended form of regular expressions, with the common operators ?, $*$, $+$, and $|$. An additional operator $\parallel$ represents the *interleaving* of the languages specified by the respective sub-expressions.

Example 1. Consider the PFA depicted in Figure 1. The node n_1 is the start node, while nodes n_2 and n_4 are final nodes. The figure shows a hypergraph representation, where the hyperedge labeled 'λ' is a *parallel* transition that does not consume any symbol. The two sub-automata represented by n_2 as well as n_3 and n_4 may proceed independently. The PFA thus accepts the language $a*\ (b* \parallel c)$, where the sub-expressions $b*$ and c are processed independently from each other.
The automaton rejects the word $''abb''$, because the non-final node n_3 is activated after consuming the input symbol 'a' but never deactivated thereafter. Final node n_4 is never activated and thus no final state is reached. The word $''abbc''$ is, however, accepted. In terms of the automaton's execution this word is equivalent to any other interleaving, e. g., $''abcb''$ or $''acbb''$.

IV. Processor Execution Modeling

Initially, lets assume that the processor, for which an execution model is to be developed, is composed by a set of *resources*, each of which is able to process one instruction at a time. Furthermore, assume that instructions traverse a subset of the resources during their execution - note that it is possible that one instruction uses several resources at the same time. In the following, we will show how parallel automata can be used to model the execution state of the processor, its resources, and its instructions.

A. Modeling Resources

A resource is modeled using four kinds of nodes in the automaton: (1) a *ready* node, (2) an *accept* node, (3) a *complete* node, and (4) two *synchronization* nodes. A local

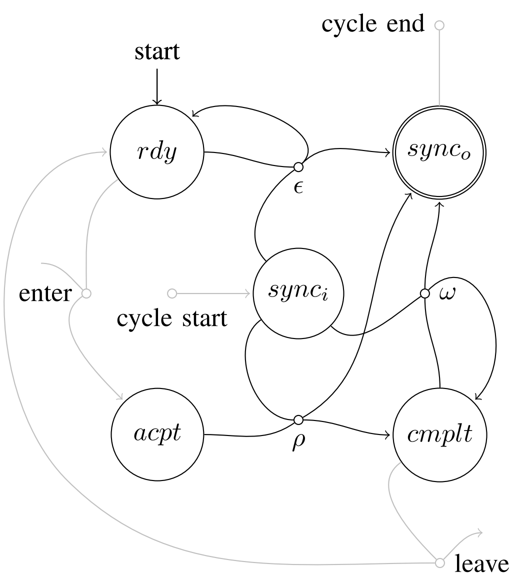

Fig. 2. Nodes and transitions representing the state of a resource in the parallel automaton.

view of the nodes and transitions of a resource is shown in Figure 2, the interaction with other automata of the execution model is indicated by the gray transitions.

The synchronization nodes separate transitions *logically* belonging to one execution cycle from those of the next cycle. The synchronization node at the center of Figure 2 serves as a token that ensures that only a single action is performed by a resource on every cycle. The activation of the second synchronization node on the other hand indicates that the resource has performed its action for the current cycle and is ready to proceed to the next cycle. The gray transactions leading to and from those nodes represent a synchronization among all the resources of the processor.

The other nodes represent the current status of the resource itself. The ready node is initially active and indicates that the resource is currently ready to accept instructions for execution. The node is deactivated whenever an instruction is accepted for execution by the resource, and may only be reactivated when the currently executing instruction has completed. When the resource is idle a transition may immediately re-activate the ready node – as indicated by transition ϵ. If an instruction has been accepted, it proceeds by performing the computation represented by the transition labeled ρ leading from the accept node to the complete node. Finally, when an instruction is blocked, i. e., it cannot proceed to another resource due to a hazard, it has to stall. This is represented by transition ω, leading from the complete node back to itself. Note that all these *resource-local* transitions disable the $sync_i$ node and in turn enable $sync_o$.

B. Instruction Execution

The sub-automata of the individual processor resources are chained together using *inter-resource* transitions as indicated by the *enter* and *leave* transitions in Figure 2. These transitions

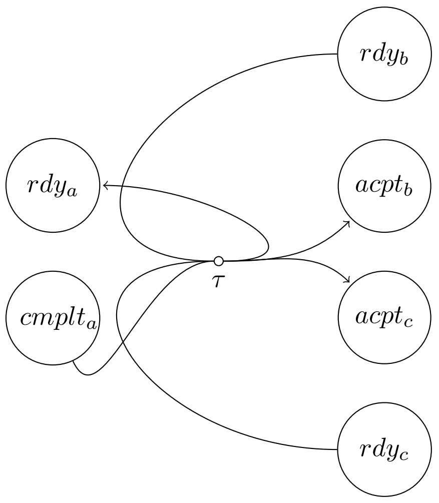

Fig. 3. Nodes and transition modeling a connection among multiple resources of the processor.

model the movement of instructions through the datapath and in addition perform arbitration of the involved resources.

Figure 3 depicts an inter-resource transition modeling an instruction that after completing its execution at resource a moves on to resources b and c, which then process the instruction in parallel. The transition is enabled only when the complete node of the source resource is active and at the same time both ready nodes of the targets are active too. When the transition is executed, the source resource becomes ready and at the same time the target resources leave the ready node and transition into the accept nodes. This effectively models how an instruction advances through the datapath from resource to resource. Furthermore, mutual exclusive use of resources is guaranteed, since the ready nodes of the target resources are properly deactivated when an instruction is accepted for execution.

C. Cycle Synchronization

Every resource is allowed to perform a single resource-local transition per cycle, since all resource-local transitions require the $sync_i$ node to be active. The $sync_i$ node thus logically marks the beginning of a cycle. Similarly, all resource-local transitions activate the $sync_o$ node, which logically marks the end of a cycle in the view of a resource. Once all $sync_o$ nodes of all resources are active a global *synchronization* transition is enabled, which re-activates all $sync_i$ nodes and signals the transition from one execution cycle to the next.

Furthermore, advanced arbitration schemes can be realized using the synchronization nodes by keeping the $sync_i$ nodes of certain resources inactive and selectively enabling them when appropriate. This can be particularly valuable in pipelined processors where resources late within the pipeline control the behavior of other resources that appear earlier in the pipeline, as for example for data by-passing and other forms of data and control hazards.

D. Transition Guards

The model presented so far is already able to express datapaths with *structural* hazards. In the presence of data or control dependencies, the transitions have to be augmented with guard functions. Similarly, dynamic scheduling of modern superscalar processors and shared resources might require additional guard functions. The guard functions inspect variables that are associated with the sub-automata of individual resources. These variables represent control decisions, data, or input operands processed by the instructions currently executed by the respective resources. The structure of the guard functions is highly dependent on the application scenario an execution model is applied to. However, our model generally simplifies the guard functions since the synchronization nodes and transitions provide a consistent view of the currently active resources and instructions.

V. Instruction-level Models

An interesting property of the approach presented in the previous section is that an instruction is represented as a set of active nodes in the automaton throughout its execution. The *movement* of an instruction through the datapath is resembled by the deactivation and activation of nodes by corresponding transitions.

It is thus possible to deduce a sub-automaton consisting of only those nodes that are potentially activated during the execution of an instruction. The result is an execution model of the given instruction, which is considerably smaller than the full processor model. This idea can also be extended to pairs of instructions or more general groups of instructions, such as all arithmetic instructions or all floating-point instructions of a processor.

VI. Discussion and Conclusion

PFAs offer a number of advantages for the modeling of processors. The processor itself is inherently parallel, so the modeling with PFAs is quite natural and easy to understand. PFA-based processor models are very visual and can be organized in order to match common graphical representations of processors such as pipeline diagrams. The movement of instructions through the pipeline is explicitly visible, which simplifies the development and debugging of PFA models.

A major advantage of PFAs over FSAs is the decoupling of states and nodes. PFAs thus consist of fewer nodes compared to equivalent regular FSA modeling the same processor. Still, the PFA implicitly encodes the same number of states. The expressive power does not change since PFAs and FSAs are equivalent. The implicit encoding of all possible processor states also simplifies the construction of PFA models, i.e., it is no longer required to pre-compute all possible states to construct the automaton.

A. Application Scenarios

The presented approach was initially intended as a flexible and generic execution model for instruction set simulators. Interpreting simulators can immediately reuse the PFA by associating its transitions with simulation functions that update the emulated processor state. In comparison to interpreters based on FSAs this will most likely induce a performance hit, since multiple transitions are required in order to simulate a single execution cycle. However, in the context of compiling simulators, which are orders of magnitudes faster than interpreting simulators, this additional information has the potential to unveil optimization opportunities, e.g., by eliminating unused code via tracing [12].

In addition to instruction set simulation, PFA-based execution models might also prove valuable for the development of verification and analysis tools. With PFAs it is possible to derive sub-automata of individual instructions, pairs of instructions or groups of instructions. These sub-automata are much smaller than the original automaton of the processor's full execution model. This opens the possibility for formal verification methods at the instruction level to proceed in two steps. The analysis first operates on the smaller instruction-level sub-automaton in order to detect possible fault conditions, such as data loss or race conditions. During the second phase the partial results obtained during the first phase are applied to the full execution model and further extended to cover the state of the complete processor. This reduces the number of processor states that have to be examined during the analysis considerably.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006.

[2] P. D. Stotts and W. Pugh, "Parallel finite automata for modeling concurrent software systems," *J. Syst. Softw.*, vol. 27, no. 1, pp. 27–43, 1994.

[3] P. Mishra and N. Dutt, *Processor Description Languages*. Morgan Kaufmann Publishers Inc., 2008, vol. 1.

[4] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, 2002.

[5] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Computer Society, 2003, pp. 556–561.

[6] F. Brandner, D. Ebner, and A. Krall, "Compiler generation from structural architecture descriptions," in *CASES '07: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 2007, pp. 13–22.

[7] F. Brandner, "Compiler backend generation from structural processor models," Ph.D. dissertation, Institut für Computersprachen, Technische Universität Wien, 2009.

[8] J. A. Bondy and U. S. R. Murty, *Graduate texts in mathematics - Graph theory*. Springer, 2007, vol. 244.

[9] M. Reshadi and N. Dutt, "Generic pipelined processor modeling and high performance cycle-accurate simulator generation," in *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Computer Society, 2005, pp. 786–791.

[10] S. Thesing, "Safe and precise WCET determination by abstract interpretation of pipeline models," Ph.D. dissertation, Naturwissenschaftlich-Technische Fakultät der Universität des Saarlandes, 2004.

[11] W. Damm and A. Pnueli, "Verifying out-of-order executions," in *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*. Chapman & Hall, Ltd., 1997, pp. 23–47.

[12] F. Brandner, "Precise simulation of interrupts using a rollback mechanism," in *SCOPES '09: Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2009, pp. 71–80.