Single-Threaded Formal Processor Models: Enabling Proof and High-Speed Execution

David Hardin[1], David Greve[1], Matthew Wilding[1], and John Cowles[2] *

[1] Advanced Technology Center
Rockwell Collins
Cedar Rapids, IA 52498 USA
{dshardin,dagreve,mmwildin}@collins.rockwell.com
[2] University of Wyoming
Computer Science Department
Laramie, WY 82071 USA
cowles@uwyo.edu

Abstract. Formal verification can be integrated into conventional computer system development by reasoning directly about system simulators. However, such simulators must be extremely fast to be usable in a real development effort. We have developed a methodology for the construction of single-threaded processor models that support both formal analysis and efficient execution. This approach can, in principle, be used with any theorem prover. We describe an instantiation of our approach for models written in the logic of ACL2, and demonstrate its scalability by constructing an executable formal model of the Rockwell JEM1 Java microprocessor. We confirm that this model is single-threaded using a tool we have developed. This check permits the safe use of destructive state update optimizations that bring the performance level of the compiled formal model to that of a conventional simulator written in C.

1 Introduction

Processor simulators are routinely built as a part of computer system development, as they allow the development of different parts of the system in parallel while providing higher performance, albeit at some cost in fidelity. For example, an executable model of the microarchitecture of a microcoded processor allows simultaneous microarchitecture implementation and microcode development. Processor models have also been built in order to support formal computer system analysis. Most of these kinds of processor models have been created after the processor is built in order to establish the correctness of the processor design, support symbolic simulation, or to verify software for the processor to execute [2, 3, 6, 10, 8, 15, 16]. Typically, different models are built to support analysis and simulation, as formal models are not often executable, much less high-performance.

* Work conducted while the author was on sabbatical at Rockwell Collins.

We believe that the unification of simulation and analysis models has advantages. Most obviously, it is easier to develop and maintain fewer models. An easily overlooked issue that arises in formal verification work is processor model validation, which is crucial to ensure that formal analysis applies to the actual machine. With a unified model we can informally validate the formal model through its use by developers as a simulator.

1.1 Executable Formal Models

Various verification projects have used theorem provers to analyze computer system models [2, 3, 5, 6, 15, 16, 8, 22], and the use of theorem provers as digital design tools is an active area of research [12]. What is not as clear is whether the languages about which theorem proving systems reason can also be used to support simulation. Execution must be extremely fast – similar in speed to the models that are currently used – and the language must be capable of integration with debugging and simulation environments in order to become part of a real development process.

Some theorem proving systems can evaluate ground functions on constants. For example, two experimental additions to the PVS theorem prover [18] allow evaluation of these kinds of terms. One prototype PVS extension is a symbolic evaluator that uses the PVS theorem prover decision procedures in a relatively automatic manner; this significantly speeds evaluation relative to conventional theorem proving strategies but is still substantially slower than conventional program execution [19]. A second prototype PVS extension is a translator of PVS functions into Common Lisp, and preliminary results indicate that execution speeds are within an order of magnitude of the speed of a model written in a conventional programming language [20]. The Nqthm theorem proving system [4] also allows relatively fast execution of its Lisp-like logic on constant terms.

Execution speeds needed for production-quality simulators can best be obtained by expressing the model in a real programming language where execution efficiency is of central importance. In this paper, we express a single model for analysis and simulation in the logic of the ACL2 theorem proving system [13]. ACL2 provides a theorem proving environment for machine-checked formal analysis and its logic is an applicative subset of Common Lisp. In the sections to follow, we describe how we overcome the inefficiencies normally associated with functional program execution to build formally analyzable computer system models that run very fast.

2 Efficient Model Execution

2.1 A Need for Speed

How fast must a processor model execute for simulation needed in processor development work? For our purposes a model must suffer no more than a 20% degradation in performance compared to our present tools in order to be usable.

In a previous study of a simple processor model written in ACL2 [23], we discovered that the unoptimized ACL2 was 130 times slower than a model written in C. Successive optimizations yielded performance that was within 10% of the C model, a very acceptable result. However, the destructive update optimizations reported in [23] were not fully justified, as the model was not shown to be single-threaded.

In this section we present several optimizations and discuss how they support faster model execution without sacrificing model analyzability.

2.2 Model Optimizations

It is possible to model the overall operation of any hardware device using only primitive logical and arithmetic operations over 32-bit integers and boolean conditional statements. GNU Common Lisp (GCL), the Lisp environment used in our experiments, provides an efficient implementation of almost all such operations. By efficient we mean both in time, meaning that they compile to efficient sequences of machine instructions, and in space, meaning that they are actually represented by their intuitive 32-bit counterpart in the generated machine code (as opposed to a "boxed" object).

2.3 ACL2-Supported Optimizations

ACL2 provides support for speeding execution of terms expressed in its logic in a safe manner. The subset of Common Lisp supported by ACL2 allows the use of type declarations which can be used by the compiler to improve efficiency. These declarations speed execution and generate corresponding proof obligations that ensure that the optimizations are made correctly.

Consider a typical term that may be found in a processor model:

```
(> n 0)
```

ACL2 normally assumes arbitrary precision integers which execute less efficiently than the native 32-bit operations provided by most host machines. We can restrict the values of n and 0 to be representable by 32 bits by replacing this term with

```
(> (bv32 n) (bv32 0))
```

where `bv32` is the following Common Lisp (and ACL2) macro:

```
(defmacro bv32 (x) `(the (signed-byte 32) ,x))
```

These declarations have two effects. First, the definition is compiled using 32-bit operations rather than arbitrary precision integer arithmetic. Second, ACL2 requires we prove that `n` and 0 are representable in 32 bits. This typically requires an additional ACL2 "guard" to the function that ensures that the function is evaluated only on values that lead to this term's evaluation on values representable in 32 bits. The ACL2 theorem prover is usually able to establish this kind of property without user assistance, as it does in this case.

By providing the appropriate type declarations, we have improved the execution speed of ACL2 processor models by as much as a factor of two in benchmark tests [23]. We use the ACL2 theorem prover to prove the generated proof obligations, so the ACL2 system guarantees the correctness of these optimizations.

2.4 Representing Hardware State

Representing the state of a machine in an efficient manner is somewhat more difficult. Typically, when one considers representing the state of a hardware device in software, one envisions a global structure, array, or list containing fields which correspond directly to registers in the hardware. The representation used may vary depending on the taste of the programmer, the type of state access required, and the perceived efficiency of a given implementation. One common theme, however, is that the state need only be accessed and updated in a very controlled fashion. In fact, it is good software engineering practice to present the programmer with a set of access and update functions which operates on the state in the expected way without ever revealing to the programmer how the state is actually implemented.

In our ACL2 processor models we represent state as a list where the most primitive access function is `NTH` and the most primitive update function is `UPDATE-NTH`. However, the Application Programmatic Interface (API) to the state is a set of accessor and update functions simply defined in terms of `NTH` and `UPDATE-NTH`. It is these functions that the programmer of the "next state" function uses, for example, to access the current program counter.

It is at the level of these accessor and update functions that we make the cut between the formal model and its actual implementation in our simulator. In the logical world of ACL2, the state is implemented as a list. In the compiled version, however, the state is implemented as a C structure with a number of fields corresponding to the various registers in the processor. GCL provides a thin foreign function interface which, as a part of the Lisp compilation process, allows direct inline replacement of Lisp functions with C implementations. This convenient feature allows us to give GCL a direct mapping from each of the defined Lisp state accessor functions (such as `(GET-PC STATE)`) to an equivalent implementation of the function in C (such as `STATE.PC`). The impact of such "strength-reducing" replacements on the efficiency of the resulting code should be obvious.

Of course, being able to direct the compiler to do things it would not normally have done is inherently dangerous. Worse, perhaps, is that these compiler directives are outside of the formalization and thus cannot be proven correct. We are thus left with the burden of justifying these directives by code inspections, and simulating the resulting model and observing the results.

2.5 Applicative vs. Destructive State Update

Much of the performance gain associated with the "under the hood" optimizations described above is the use of destructive updates on a single global processor state, rather than the applicative update process, which involves much object creation and copying. This fundamental change could lead to inconsistencies, and so we look again to the theorem prover to demonstrate that the use of destructive operations is indeed safe.

Functions using destructive state updates can be shown to be equivalent to those using applicative updates if the updates are single-threaded, that is, that no updates are done while the previous state can still be accessed (i.e., is still "live").

In order to ease the programmer's task to create single-threaded models, we provide a single-threadedness checker (described in Section 3), as well as a macro form. This macro, much like ACL2's top level `PROGN` form, hides the sequential updating of the state. By employing this macro in the body of the functions we define we provide the programmer with a "correct by construction" means of operating on the state object. This macro identifies state accessors and updaters presented to it in a common form, inserts the appropriate state argument and performs the required state bindings. It also recognizes common arithmetic expressions and local variable bindings and automatically generates type constraints, via `THE` and `DECLARE` forms, to guarantee that such operators and objects are implemented as efficiently as possible.

3 Checking Models for Single-Threaded Execution

So, how does one use a theorem prover with a logic assuming applicative semantics to reason about functions implemented, for the sake of fast execution, with destructive side effects on global objects?

By way of example, suppose the contents of pairs of data items can be destructively updated by the following:

`(update-lt pair value)` replaces the left coordinate of `pair` with `value`, and
`(update-rt pair value)` replaces the right coordinate of `pair` with `value`.

Suppose further that the applicative logic of the theorem prover treats `update-lt` and `update-rt` as functions that construct new pairs. Assume the order of evaluation is described by innermost-first, left-to-right, and call by value. Suppose further:

`(left pair)` returns the left coordinate of pair, and
`(right pair)` returns the right coordinate of pair.

As an example, consider the following transformation:

```
(update-rt (update-lt pair (right pair))                    (1)
   (left pair))
```

This transformation in the logic differs from that produced by the destructive implementation. In the logic, a new pair is returned with the coordinates of the original pair in reverse order. However, if a pair is destructively updated, the resulting pair has both coordinates equal to the right coordinate of the original pair, because the `update-lt` overwrites the original left coordinate of pair before it can be extracted.

We have developed a tools which checks two properties of transformations to ensure that the the implementation is accurately reflected in the logic:

1. The transformation must compute the same results whether using destructive side effects on the global objects or using applicative semantics.
2. Global objects must not be updated by any function other than the primitive update operations or previously defined transformations that satisfy these two properties.

Property 1 is checked by syntactically reformulating the proposed definition of an updating transformation into a new definition that is faithful to the semantics imposed by destructive updates. This is done using nested `LET`'s to explicitly control the order of the bindings, and by carefully not creating any aliases for global objects. The ACL2 theorem prover is then used to establish that for every input the two definitions return equal results. This is proved in a very limited theory, essentially consisting of the logic plus the two definitions.

For (1), this reformulation yields the transformation:

```
(let ((v$3 (right pair)))
     (let ((pair (update-lt pair v$3)))
          (let ((v$6 (left pair)))
               (update-rt pair v$6))))
```

The theorem prover *fails* to prove that this transformation computes the same results as (1). At this point, (1) is conservatively rejected because its logical semantics *may* not match its destructive implementation.

If (1) is rewritten to temporarily save the original left coordinate, then the result computed in the logic always matches the result returned by the destructive implementation:

```
(let ((temp (left pair)))                                   (2)
     (update-rt (update-lt pair (right pair))
                temp)))
```

Reformulating (2) yields

```
(let ((v$2 (left pair)))
     (let ((v$6 (right pair)))
          (let ((pair (update-lt pair v$6)))
               (update-rt pair v$2))))
```

which the theorem prover verifies computes exactly the same results as (2).

Property 2 is checked by ensuring that any function or transformation that uses a global object as input is among the functions and transformations with permission to do so. In this example, `update-lt`, `update-rt`, `left`, and `right` would have the right to use global pairs as inputs, so (2) has this property. Thus, (2) is single-threaded, and destructive update optimizations may be used.

3.1 Related Work

It would be better to integrate checks similar to these into the theorem prover. Inspired in part by some of our previous work [23], we know of two efforts, one involving the PVS theorem prover and the other the ACL2 theorem prover, to safely allow some implementations of applicative functions using destructive global objects.

PVS [18] has been experimentally augmented to perform a static analysis, based on [9], to determine when destructive updates may be safely used [20]. So it is possible that within the definition of any given function some updates may be done destructively while others are done nondestructively; each primitive update function may have a destructive implementation as well as one that is not destructive.

The internal state of the ACL2 theorem prover is already implemented with destructive updating. Severe syntactic restrictions are imposed to ensure that the applicative semantics assumed by the logic agrees with the destructive side effects of the implementation. The creators of ACL2 have publicly committed [17] to providing automatic enforcement of these syntactic restrictions on user designated objects as well.

4 Application to the JEM1 Microprocessor

We have applied the techniques described in Section 2 and Section 3 to the construction of analyzable and speedy microarchitectural models for our production microprocessor designs. The Rockwell JEM1 [11, 24] is the first microprocessor to directly execute Java Virtual Machine bytecodes [14]. We have constructed a complete microarchitectural model of the JEM1 in ACL2, constituting some 13,000 lines of Lisp. In this section we describe how a part of the JEM1 processor model was written in order to allow it to be both reasoned about and used to support mainstream development.

4.1 An Example

Consider the `CLEAR_REG_SELECT` function, which clears a range of JEM1 register selectors, as shown in Figure 1.

This function illustrates several features of the model. First, state-updating is done only through a few allowed operators (e.g., `CURR.PUT-REGSEL`). Type declarations are used to prevent "boxed" objects, and to provide hints to the

```
(DEFUN CLEAR_REG_SELECT (INDEX ST)
  (DECLARE (TYPE (SATISFIES JEMST) ST) (TYPE (INTEGER 0 33) INDEX)
           (XARGS :MEASURE (NFIX (- 33 (NFIX INDEX)))))
  (LET ((*ST0 ST))
    (IF (THE (SATISFIES BOOLEANP)
          (TRUE (INT-GT<32> (NFIX INDEX) (REGNUM))))
        *ST0
        (LET ((*ST1 (CURR.PUT-REGSEL (C.FALSE) INDEX *ST0)))
          (CLEAR_REG_SELECT (+<32> 1 (NFIX INDEX)) *ST1)))))
```

Fig. 1. Portion of ACL2/Common Lisp JEM1 model

compiler to use efficient machine data types wherever possible. Additionally, special arithmetic and logical operator macros such as +<32>, INT-GT<32> are utilized such that, for example, +<32> expands to an addition function that is declared to have 32-bit arguments and return value. Nested LET's (automatically generated by a macro form that has been expanded here for clarity) make this function "single-threaded by construction", and thus easily passes the single-threadedness check of Section 3.

The reader will also note that this particular function is recursive, as this is the only way to achieve iteration in ACL2. However, since CLEAR_REG_SELECT is properly tail-recursive, the GCL compiler optimizes the recursive call away, resulting in an efficient loop-with-conditional-branch structure in the generated machine code.

4.2 Results

Following the approach given in Section 2, we sped the Common Lisp execution of the JEM1 model by replacing the access and update functions with fast, destructive versions, once the single-threadedness check described in Section 3 had been successfully completed for the 40 functions in the JEM1 model which update the processor state.

Additionally, we have integrated the formal JEM1 model into a production development environment in order to measure its speed and demonstrate its ultimate practicality. The JEM1 development tools allow the execution of Java code on either the JEM1 hardware or a JEM1 simulator. We substituted the ACL2-based JEM1 simulator for the current C-based simulator, integrating it with the Averstar JWatch Java debugger [1] frontend. Thus, debugging Java code using this tool exercises our formal model, utilizing an easy-to-use graphical debugger interface. The user is blissfully unaware that a formal model is being used, as our optimized JEM1 model runs very fast, *indeed as fast as the original C model* in our benchmark tests.

We have also used *the same model* to prove theorems about the JEM1 design. In particular, we have proved that the overflow bit generated by the ALU is set exactly when the ALU result is unrepresentable by a 32-bit signed value. The proof of this lemma using the ACL2 theorem prover required less than a man-week of effort, and benefited from a previously-constructed library of ACL2 theorems related to hardware [7].

5 Conclusions and Future Work

Overcoming the slow execution speed of applicative models is crucial in order to use a single model for both formal analysis and simulation. The optimizations described in this paper, featuring the replacement of applicative state update functions with destructive versions, provide speed equal to that of an informal model written in C, but could give inaccurate results if used in a multi-threaded manner. Thus, we have created our own single-threadedness check for ACL2 models, which allows us to use destructive versions of state updaters safely. We have applied this method to a real microprocessor design, establishing that the approach scales to industrial problem domains, and demonstrating that the resulting model is both fast and integrable into a conventional engineering development environment.

The developers of the ACL2 and PVS theorem provers are currently adding single-threadedness checks into their tools; eventually we will be able to dispense with our own checker in favor of one of these integrated ones. We think this capability will be of great utility to the theorem proving community, and we are eager to explore areas other than digital hardware design where the existence of such checks can be exploited to improve performance without invalidating claims of correctness.

Moreover, we plan to explore further how formal processor models can improve our computer system development work. In the short term, we are slated to build executable formal models of two new processor designs, as well as a peripheral chip. In the medium term we plan to integrate symbolic simulation results into our design process and use symbolic results to aid our regression testing [10]. In the long term we plan to verify large portions of our designs using theorem provers.

References

1. Averstar Inc. JWatch home page, 1998. http://www.jwatch.com.
2. William R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.
3. William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
4. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.
5. Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

6. Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
7. Bishop Brock. ACL2 integer hardware specification (IHS) books, 1998. (available on ACL2 home page).
8. Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
9. M. Draghicescu and S. Purushothaman. Uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, 1993.
10. David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
11. David A. Greve and Matthew M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, January 12, 1998.
12. David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
13. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203 – 213, April 1997.
14. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts, 1996.
15. Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.
16. Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.
17. J Moore. Personal Communication, November 1998.
18. S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
19. Stephan Pfab. Personal Communication, November 1997.
20. John Rushby. Personal Communication, October 1998.
21. Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356, 1997. (http://atb-www.larc.nasa.gov/Lfm97/).
22. Matthew M. Wilding, David A. Greve, and David S. Hardin. Efficient simulation of formal processor models. Technical report, Rockwell Collins Advanced Technology Center, October 1998. Unpublished manuscript.
23. Alexander Wolfe. First Java-specific MPU rolls. *Electronic Engineering Times*, page 1, September 22, 1997.