Scalable Hybrid Verification of Complex Microprocessors

Maher Mneimneh, Fadi Aloul, Chris Weaver, Saugata Chatterjee, Karem Sakallah, Todd Austin
University of Michigan
{maherm, faloul, chriswea, saugatac, karem, taustin}@umich.edu

ABSTRACT
We introduce a new verification methodology for modern microprocessors that uses a simple checker processor to validate the execution of a companion high-performance processor. The checker can be viewed as an at-speed emulator that is formally verified to be compliant to an ISA specification. This verification approach enables the practical deployment of formal methods without impacting overall performance.

1. INTRODUCTION
Modern microprocessors are enormously complex systems. Architectural features, such as out-of-order and speculative execution, branch prediction, and multi-level caches, significantly enhance performance but create serious functional verification challenges. By far, the most common verification paradigm in use today is simulation. A typical microprocessor design is simulated for months using a battery of test programs that are intended to "wring the bugs out" before the design is released for manufacture. Such a verification strategy is becoming increasingly untenable, however, because of increasing design complexity and time-to-market pressures. In addition, simulation-based verification cannot guarantee that a design is bug-free; at best, simulation can reduce the probability of releasing a buggy design to acceptable levels but can never completely certify design correctness.

In this paper, we present a new verification methodology that significantly lowers the burden of verifying modern microprocessor designs. The verification process is decomposed into two steps: a) dynamic on-line verification of the *core processor* P_{core} by a companion *checker processor* P_{check} and b) static off-line formal verification of P_{check} with respect to the instruction set architecture (ISA) specification. P_{check} resides in the final "commit" stage of P_{core} and dynamically verifies P_{core}'s execution of each instruction before retiring it to architected storage (register file and memory.) P_{check} operates in two modes: a *check mode* in which it concurrently checks the computation, communication and control information generated by P_{core}, and a *recovery mode* which it enters when the check mode indicates potential errors in the data it is receiving from P_{core}. In such cases, P_{check} stalls P_{core}, re-executes the "faulty" instruction, and then re-starts P_{core} and resumes operating in its check mode. The correctness of P_{check} is established formally by comparing its implementation to a specification that is extracted from the corresponding ISA reference manual. The comparison is performed for every instruction (or instruction type) and yields a formula in the *Logic of Equality with Uninterpreted Functions* (LEUF) that is subsequently checked for validity.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

The viability of such a hybrid verification approach rests on the availability of a checker processor that is both fast and simple. It should be fast so that it doesn't degrade the performance of P_{core}, and simple so that its formal verification is tractable. We show in this paper that it is indeed possible to design such a processor and show how we plan to formally verify it.

The paper is organized as follows. In Section 2 we review relevant work in formal microprocessor verification and show how it relates to the particular approach we describe in this paper. Section 3 is devoted to the description of dynamic verification; in particular, we describe the architecture of a proposed checker processor and show that it does not lead to a degradation in the performance of the core processor. Section 4 briefly reviews the logic of equality with uninterpreted functions, its use in hardware verification, and how the validity problem in this logic is converted to a Boolean satisfiability (SAT) problem. In Section 5, we describe our methodology for verifying the checker processor against the ISA specification and illustrate it with an example. Experimental verification results for a prototype checker are discussed in Section 6 and the paper is concluded in Section 7 with some pointers to future work.

2. PREVIOUS WORK
Formal microprocessor verification methods represent an alternative to traditional verification by simulation. Formal verification establishes a relation between a microprocessor implementation and its specification. The first attempts were aimed at specifying a simple microcoded computer and formally verifying its RTL implementation using HOL [12]. In HOL, the user formulates the theorems to be proved and provides guidance to the mechanical proof system. Thus, using the system requires a great deal of expertise. The authors reported that the verification procedure required six hours of run time and about two man-months of effort to construct the HOL equations. Another early effort was the attempt to verify the FM8501 [11], a computer similar in complexity to a PDP-11. The specification was described at the instruction set level and the implementation at the register transfer level. The Boyer-Moore theorem prover was used to show that, after the execution of a macroinstruction in the specification and the corresponding N microinstructions in the implementation, corresponding machine states in both models were equivalent. The reported advantage of the Boyer-Moore theorem prover was its use of heuristics that speed up the proof process and that require less user expertise than HOL.

Recently theorem provers [15] were used to verify a pipelined microprocessor. The microprocessor was described at two levels: an *abstract specification* which captures the semantics of the instruction set and an *abstract realization* which models the activities that occur in every clock cycle. The correctness properties were constructed and proved interactively. The authors indicated that the verification task took about four man-months by an expert user of the theorem prover. The PVS theorem prover was also used to establish the correctness of another pipelined microprocessor [9]. The implementation and specification were described using state machines and the correctness criteria were based on showing that traces generated by the two machines are the same. Timing differences between the specification and implementation were accounted for either by "slowing down" the specification state machine or "speeding up" the implementation state machine.

In all of the above methods, human interaction was an essential factor guiding the proof of correctness. In addition, the processors verified were simple compared to current state-of-the-art processors. The method proposed by Burch and Dill [7] greatly enhanced verification techniques. By using an automated decision procedure, correctness proofs required less human interaction and made possible the verification of complicated designs that previous methods failed to handle. Their method is composed of two phases. In the first phase, an implementation is extracted from the RTL description and modeled in a Lisp-like language. Similarly, the corresponding specification is extracted from the ISA description and modeled using the same language. These compiled models of the specification and implementation are then used to construct a formula that is correct if and only if the implementation correctly realizes the specification. In the second phase, the validity of the constructed formula is checked. Proving the validity of the generated equation is an automated task. This significantly reduces the burden of verification when compared to the mechanical proofs conducted in previous methods. The logic they used is the quantifier-free logic of uninterpreted functions with equality and propositional connectives. This logic proved to be effective in verifying microprocessors because it allows the abstraction of the processor datapath by using uninterpreted functions. In addition, the validity problem in this logic is decidable. Although their method was able to verify a DLX implementation, they report that their decision procedure presented a bottleneck when applied to commercial processors.

More efficient procedures for the logic of equality with uninterpreted functions were proposed by Bryant et al. [6]. These procedures exploit the structure of the formulas generated to reduce the complexity of the decision procedure for the logic. By distinguishing between positive and negative terms, proving the universal validity of a positive formula can be reduced to proving its validity in all maximally diverse interpretations. Thus, the decision procedure can significantly reduce the different interpretations it should consider. Their method was able to efficiently verify complex pipelined microprocessors that defied all previous methods.

The technique we present in this paper for formally verifying the checker processor borrows from the previous work described above. Yet, it has several distinct advantages, the most significant being its scalability. The complexity of microprocessor designs tends to grow rapidly with the introduction of advanced microarchitectural features. Although recent methods were able to verify some of these modern designs, the techniques used are not guaranteed to perform efficiently on future designs. Our approach scales well as designs get more complex since the architecture of the checker processor will not drastically change. In addition, our approach relaxes the requirement on the specification. Whereas previous techniques required an "executable" specification, our approach uses an "interpretation" of the ISA reference manual as a specification.

3. DYNAMIC VERIFICATION

Dynamic verification [4] is an online instruction checking technique that stems from the simple observation that *speculative execution is fault tolerant*. Consider, for example, an incorrectly designed branch predictor that indexes the predictor array with the most significant bits of the program counter (instead of the least significant bits.) The design would operate correctly even though the branch predictor contained a design error. The only effect on the system would be significantly reduced branch predictor accuracy (*i.e.*, more mispredictions) and accordingly reduced system performance. From the point of view of a correctly-designed branch predictor check mechanism, a bad prediction from a broken predictor is indistinguishable from a bad prediction from a correct predictor design.

Given this observation, the burden of verification in a complex design can be decreased by simply increasing the degree of speculation. Dynamic verification does this by pushing speculation into all aspects of core program computation, communication, and control. Accordingly, a robust speculation check mechanism will ensure that permanent or transient faults do not impact program correctness. Figure 1 illustrates the approach.

To implement dynamic verification, a microprocessor is constructed using two heterogeneous internal processors that execute the same program. The complex *core processor* P_{core} is responsible for pre-executing the program to create the *prediction stream*. The prediction stream consists of all executed instructions (in program order) with their input values and memory addresses referenced. The core processor is identical in every way to a traditional complex microprocessor core up to (but not including) the retirement stage. The core processor is "predicting" values because it may contain latent design errors that could render some instruction results incorrect.

The simple *checker processor* P_{check} follows the core processor, verifying the activities of the core processor by re-executing all program computation in its wake. The high-quality stream of predictions from the core processor serves to simplify the design of the checker processor and speed its processing. In the event the core produces a bad prediction value (*e.g.*, due to a design error), the checker processor will fix the errant value and flush all internal state from the core processor, and restart it after the errant instruction. Once restarted, the core processor will re-synchronize with the correct state of the machine as it reads register and memory values from non-speculative storage. The resulting dynamic verification architecture should benefit from a reduced burden of verification, as only the checker need be completely correct. Since the checker processor will fix any errors in the instructions that are to be committed, the verification of the core is reduced to locating and fixing commonly occurring design errors that could affect system performance.

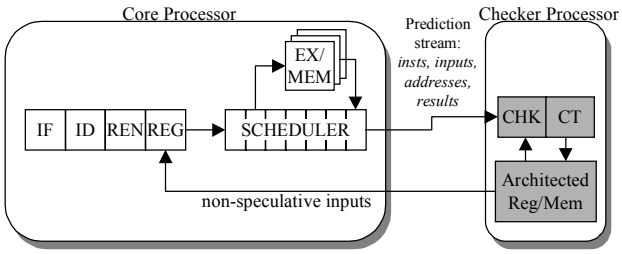

Figure 1: Dynamic Verification Architecture

In normal checking mode, when the core processor retires an instruction, the checker pipeline receives the instruction with core processor predictions. These predictions include the next program counter, instruction, inputs to the instruction, and memory addresses referenced (for loads and stores.) Using these high quality predictions, the checker processor re-executes in parallel the four fundamental steps of instruction execution: fetch, decode, execute, and memory access. These processing steps, which ordinarily execute serially, proceed in parallel in the checker processor because the core processor predictions break dependencies that exist between these steps. If each prediction from the core processor is determined to be correct, the result of the current instruction (a register or memory value) as computed by the checker processor is allowed to retire to non-speculative storage in the *commit* (*CT*) stage of the checker processor. In the event any predictions from the core are found to be incorrect (*e.g.*, due to a design error), the errant result is fixed by the checker and the core processor is flushed and restarted.

Instruction recovery is implemented by reconfiguring the checker pipeline into a serial unpipelined processor, similar to the classic five-stage pipeline. In this mode, stage computations are sent to the next logical stage in the checker processor pipeline, rather than used to verify core predictions. Unlike the classic five-stage pipeline, only one instruction is allowed to enter the pipeline at a time. As such, the recovery pipeline configuration does not require bypass datapaths or complex scheduling logic to detect hazards. Once the instruction has retired, the checker processor re-enters normal processing mode and restarts the core processor after the errant instruction. An important aspect of the checker design is that the check and recovery modes use the same checking modules, thereby reducing the area cost of the checker and its design complexity.

Pre-execution of the program on the complex core processor eliminates most of the processing hazards (*e.g.*, branch mispredictions, cache misses, and data dependencies) that could slow the simple checker pipeline. A recent paper [8] used detailed cycle-based simulation to gauge the performance impacts of the checker processor. For a large collection of benchmarks including programs from the SPEC benchmark suite, slowdowns due to instruction checking were at most 0.2%. The study found that very few checker pipeline stalls existed. Branch hazards were cleared by the high quality next-PC predictions from the core processor, generated during pre-execution of the program. Data hazards from long latency instructions were eliminated by the input value predictions supplied by the core processor. With these values, instructions need not wait for earlier dependent operations to finish. Finally, data hazards from cache misses were virtually non-existent in the core processor because the core processor ran ahead of the checker initiating cache misses (and tolerating their latency) in advance of checker execution. An earlier study [4] found that while recovery mode performance was quite poor, overall slowdown were less than 1% if fault rates were limited to at most one fault per 1000 cycles. A moderate level of core processor verification should be sufficient to locate and fix frequent design errors that could adversely impact system performance.

Besides reducing the burden of verification in complex microprocessor designs, dynamic verification may render other benefits and opportunities in the design of complex microprocessors. A number of promising directions have been suggested (additional details are available in [4].) These proposals include beta-release processors to reduce time-to-market and design cost, SER and transient fault tolerant designs, aggressive core circuitry implementations, and designs that reduce core processor complexity.

4. DATAPATH/MEMORY ABSTRACTION

Microprocessor verification is typically split into independent datapath and control logic verification tasks. This naturally leads to a hierarchical verification methodology wherein datapath and memory subsystems are verified separately and are assumed to be correct for purposes of verifying the control logic. In most modern formal verification approaches for control logic, the quantifier-free logic of equality with uninterpreted functions (LEUF) [7] provides a convenient formalism for datapath and memory abstraction.

Using this logic, an instruction such as "ADD R1, R2, R3" which adds the integer contents of registers R1 and R2 and stores the result in register R3 would be represented by the LEUF formula "v3 = A(v1, v2)" where v1, v2, and v3 are symbolic term variables that denote the contents of their corresponding registers, and A is the function symbol for the uninterpreted integer ADD function. Note that this type of abstraction makes the width of the datapath transparent to the verification process, significantly reducing its complexity without affecting its correctness.

Efficient algorithms exist for checking the validity of formulas in this logic. In particular, the Stanford Validity Checker, SVC [5], has been successfully used to prove the correctness of formulas generated for pipelined microprocessor verification. An alternative approach for checking the validity of LEUF formulas is based on converting them to propositional logic and testing them for satisfiability using a Boolean SAT solver. In general, this conversion results in an exponential increase in the size of the formula and is impractical except in certain special cases. In our case, as we show in Section 6, this conversion did not present an obstacle and proved to be very effective due to the simplicity of P_{check}.

LEUF formulas are typically converted to propositional form by instantiating new *terms* (domain variables) for each uninterpreted function occurrence, and subsequently encoding these *terms* using Boolean variables. The details of such conversions differ [6,10], but they all yield a Boolean formula that is equivalent to the original LEUF formula. One of the simplest conversion algorithms is Ackermann's method [1] which replaces uninterpreted functions in the original formula by new domain variables (as described above) and augments it with additional constraints on those variables to preserve equivalence. For example, the LEUF formula

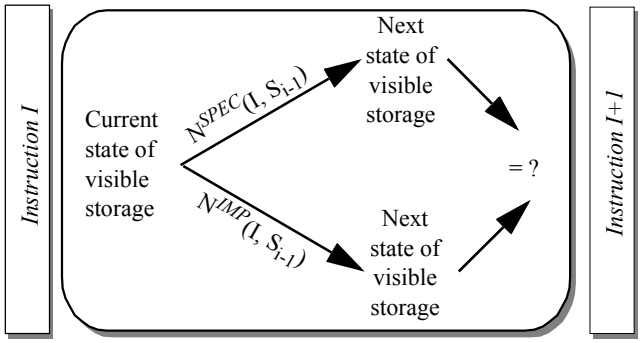

Figure 2: An overview of the verification methodology

$\alpha(A(x, y), A(u, v))$ which contains two occurrences of the uninterpreted function A with different arguments is replaced by

$$(((x = u) \wedge (y = v)) \rightarrow (fA_1 = fA_2)) \rightarrow \alpha(fA_1, fA_2)$$

where fA_1 and fA_2 are two new domain variables that denote the two separate occurrences of uninterpreted function A. To complete the conversion of the formula to propositional form, the k domain variables are suitably encoded by $\lceil \log k \rceil$ Boolean variables. This encoding follows from a theorem, by Ackermann, that the universal validity of a formula in the logic of equality is preserved when it is proved valid in a domain with k elements regardless of the domain size of the formula's variables. With such an encoding, each equality term of the form $(a = b)$ is finally replaced by

$$\bigwedge_{i=0}^{\lceil \log k \rceil} (a_i \odot b_i)$$

5. ISA VERIFICATION METHODOLOGY

Using our hybrid verification methodology, the correctness of the P_{core}/P_{check} system can now be comprehensively established by only proving the correctness of P_{check}'s implementation against the ISA specification. This verification task is considerably simpler than those proposed for verifying an out-of-order complex instruction processor against a corresponding unpipelined specification and is much more amenable to automation.

A high-level overview of this verification procedure is depicted in Figure 2. The ISA specification and the P_{check} implementation are viewed as *next-state functions* that modify programmer-visible state (memory, register file, program counter, etc.) in response to instruction execution. Most ISA specifications for general-purpose processors assume a sequential execution model in which instructions are fetched, decoded, executed, and retired to architected storage. In addition, the effect of executing any given instruction on visible state is completely determined by what that state is, regardless of how it was reached. This allows us to check the correctness of individual instructions (or instruction types) without worrying about interactions with other instructions. Another way of saying this, is that the effect of executing any instruction is completely recorded in programmer-visible state and not in any hidden control state.

Denoting the next-state functions for the ISA specification and implementation of a particular instruction I as N^{SPEC} and N^{IMP}, respectively, the verification task becomes one of insuring that $N^{IMP}(I, S_{i-1}) = N^{SPEC}(I, S_{i-1})$ where S_{i-1} represents the current state of visible storage. The complexity of this check depends primarily on the nature of N^{IMP}. A simple in-order single-issue unpipelined processor, such as P_{check}, yields a simple next-state function for each of its instructions, significantly reducing the computational requirements for verification. On the other hand, a speculative, out-of-order pipelined implementation creates inter-instruction dependencies that must be accounted for when determining an instruction's next-state function. Such a function is likely to be complex due to the presence of extra control state, and could drastically increase the complexity of verification. It is important, though, to note that even in such cases, these functions must conform to the sequential execution semantics at instruction boundaries.

Format
SUBQ Ra.rq, Rb.rq, Rc.w

Operation
Rc ⟵ Rav - Rbv

Instruction Mnemonics
SUBQ Subtract Quadword

Description
Register Rb or a literal is subtracted from register Ra and the 64-bit difference is written register Rc.

(a)

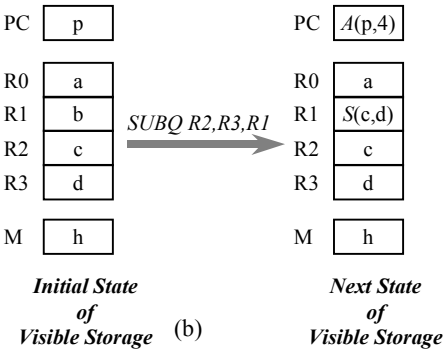

Figure 3: SUBQ instruction example (a) Specification as described in the Alpha Reference Manual (b) Effect of execution as described by the specification

The verification procedure checks for errors in the implementation of P_{check}'s control logic. Datapath components, such as the ALU, register file, and memory system are assumed to be correct and are abstracted using suitable uninterpreted functions. In the illustrations that follow we will use uninterpreted function symbols A to stand for integer addition, S for integer subtraction, *SE* for sign extension, and *READ* and *WRITE*, respectively, for reading from and writing to memory or the register file.

Our verification procedure is best illustrated by an example. Figure 3(a) shows an excerpt from the Alpha Architecture Handbook [3] describing the operation of the *quadword subtract* (*SUBQ*) instruction. We extract N^{SPEC} by a 'syntactic interpretation' of the instruction description. This step is illustrated in Figure 3(b) where we interpret the effect of this instruction on a hypothetical model having four registers and one memory location. Note that this in-

```
// Decode an instruction given the instruction register
function [6:0] decode_cmd;
      input [31:0] IR;
      case (IR[31:26])
     .....
     `INTA_GRP:
         `SUBQ_INST:decode_cmd = `SUBQ;
      ......
// Decode the value of the first operand
function [63:0] decode_RA;
      input [6:0] cmd;
      input [63:0] READB_BUSRF, READA_BUSRF, CHECKER_PC;
     case (cmd)
     ....
     default: // ALU operations
         decode_RA=READA_BUSRF;
     ....
// Decode the value of the second operand
     ....
// Decode the destination register
decode_DST
     ....
```

Figure 4: Excerpt from the Verilog code of P_{check} prototype

struction has no effect on memory states permitting us to abstract memory using a single symbolic location *M* containing the symbolic value *h*. N^{SPEC} changes the content of the program counter to *A(p,4)* and the content of the destination register *R1* to the result of the *SUBQ* instruction, *S(c,d)*. Although the Alpha Architecture Handbook specifies 32 programmable-visible registers, we are limiting this example to just 4 registers for illustrative purposes. Note, also, that we are specifying particular, as opposed to arbitrary symbolic, source and destination registers in this example, even though this is not a requirement of the verification process.

The implementation next-state function for *SUBQ* is extracted from a Verilog HDL model as shown in Figure 4 and Figure 5. The Verilog excerpt in Figure 4 highlights some of the relevant code sections that affect the control flow of this instruction. When symbolically simulated, as shown schematically by the bold lines in Figure 5, we obtain the implementation's symbolic expressions for each of the programmer-visible states.

At this point the following LEUF formula is constructed:

$$\alpha =$$
$$(p = q) \wedge (a = w) \wedge (b = x) \wedge$$
$$(c = y) \wedge (d = z) \wedge (h = j)$$
$$\rightarrow$$
$$(A(p,4) = A(q,4)) \wedge (S(c,d)) = S(y,z)) \wedge$$
$$(a = w) \wedge (c = y) \wedge (d = z) \wedge (h = j)$$

stating that if the specification and implementation start in identical initial programmer-visible states, then after execution of *SUBQ*, they should yield identical next states. For this simple example, the validity of the above formula is obvious by simple inspection. In general, though, the formula must be a) converted to propositional form as described in Section 4, b) negated, c) translated to conjunctive normal form (CNF), and finally d) checked with a SAT solver. If the solver proves the formula to be unsatisfiable, then the implementation of the instruction is correct. Otherwise, the instruction's control implementation is erroneous and the satisfying assignment returned by the SAT solver can be used to diagnose the error.

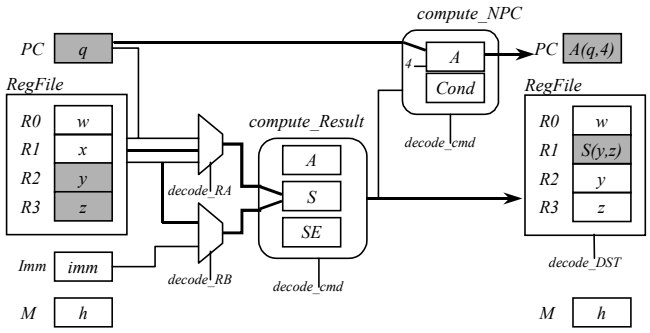

Figure 5: Symbolic simulation of SUBQ on P_{check} prototype

The above procedure must be repeated for each instruction in a processor's ISA specification, in both the check and recovery modes of the checker processor implementation. The implementation is certified to be correct if all constructed formulas are proved valid. It is important to remember, though, that this proof of correctness applies only to the control logic of the implementation. This includes control logic errors that cause invalid accesses (reads or writes) to memory or the register file, but not errors in the controllers of those storage structures. Such errors, e.g. in the address decoder of the register file, must be checked by other techniques. In particular, when verifying *load* or *store* instructions, we only check that the correct symbolic addresses are generated by the implementation's control unit but not that the correct memory locations are accessed.

It is interesting to note that our verification problem differs from those studied previously in that we check an implementation directly against the ISA specification. Previous methods [6, 7] required an unpipelined 'executable' model as the specification and checked a complex pipelined implementation against it. In our methodology, this more difficult verification problem is performed dynamically using the P_{core}/P_{check} combination whereas the simpler problem of verifying P_{check} against the ISA is handled formally.

6. EXPERIMENTAL RESULTS

Our initial evaluation of the verification methodology proposed in this paper was based on a prototype implementation of P_{check} based on the Alpha ISA [3]. The current design supports 30 instructions and consists of 2000 lines of structural Verilog code. All experiments were conducted on a 333 MHz Pentium II machine having 512 MByte of RAM and running the Linux operating system.

The verification task consisted of a "manual" symbolic simulation of the execution of each of these instructions in P_{check}'s recovery mode. This was done by tracing instruction flow through the Verilog code and monitoring changes to architected storage. An LEUF validity formula was subsequently generated and processed as described in the previous section. We used the GRASP [14] SAT solver as our satisfiability engine.

A sample of the verification results is shown in Table 1. Column 1 in the table specifies the instruction being verified, columns 2 and 3 give the number of variables and clauses in the corresponding CNF formula, column 4 indicates whether the formula was satisfiable or unsatisfiable, and column 5 shows the CPU time required for check-

TABLE 1: Experimental Results

Inst	V	C	S/U	Time, sec.
ADD	542	6430	U	0.22
LDQ	583	7285	S	0.33
BRNE	563	6430	U	0.47
BR	549	6288	S	0.28

ing the formula. As can be seen from this table, two of the instructions (*ADDQ* and *BRNE*) were found to be correct. Interestingly, though, the other two instructions (*LDQ* and *BR*) were flagged as having erroneous implementations.

The implementation of the *LDQ* instruction, which loads a value from a calculated memory location into a specified register, yielded the symbolic expression *READ(A(R2,disp))*. On the other hand, the instruction's specification from the Alpha reference manual resulted in the expression *SE(READ(A(R2,SE(disp))))*. The discrepancy between these two "next states" was discovered by the SAT solver and led to the source of the error, namely the absence of a sign-extension unit in P_{check}. It is worth noting that this type of error could be difficult to identify using simulation, since not all calculated results exercise the bug. Actually, values with a zero in their most significant bit would execute correctly in this buggy model since sign extending their value doesn't change it.

Diagnosis of the error in the unconditional branch instruction *BR* revealed that the register indicated in the instruction was not updated with the new value of the program counter as required by the ISA specification. Conceivably, this type of bug can be detected by careful simulation.

In some rare cases, the SAT solver returned a satisfiable assignment for an instruction that was correct. Such false positives resulted from LEUF specification and implementation formulas that were identical except for different orders of arguments to, as well as different orders of applications of, uninterpreted functions. Such functional properties (namely, commutativity and associativity) are not modeled in the constructed LEUF formula and must be explicitly accounted for with additional constraints.

We also discovered in this exercise some inconsistencies in the ISA specification itself. For example, the RTL description of the *LDQ* instruction clearly shows that the displacement field is sign-extended to 64 bits and then left shifted by two bits: *SL(SE(Disp), 2))*. On the other hand, the English language description of the same instruction implies that the displacement field is first shifted left and then sign-extended: *SE(SL(Disp, 2))*. Since these two operations are not commutative, one of these descriptions (the English language comment) is clearly incorrect. This observation reinforces the need, recognized in previous work on bus protocol verification [2,13], to seek formal representations of specifications.

7. CONCLUSION

We presented a design-for-verifiability technique that promises to mitigate the current functional verification bottleneck by combining online dynamic with offline formal verification methods. Noting that the semantics of instruction set processors are typically quite simple and do not grow in complexity over time, whereas the implementations of such processors continue to increase in complexity in order to achieve higher performance targets, our approach has built-in scalability that insures its ability to verify future highly-optimized processors. Our initial data suggest that this approach to design verification has little impact on performance and is computationally tractable to enable practical application on any ISA. We plan to explore the automatic derivation of the validity next-state formulas from RTL models of checker processors and to extend this technique to all instruction types and architectural features such as interrupts and exceptions. We will also investigate other approaches, besides Boolean satisfiability, for checking the validity formulas. We are currently designing a complete checker processor as a proof-of-concept vehicle for estimating area, power, and performance costs.

8. ACKNOWLEDGMENTS

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center.

9. REFERENCES

[1] W. Ackermann, "Solvable Cases of the Decision Problem," Noth-Holland, Amsterdam, 1954.

[2] F. A. Aloul and K. A. Sakallah, "Efficient Verification of the PCI Local Bus using Boolean Satisfiability," in IWLS, 2000.

[3] "Alpha Architecture Handbook," Product #EC-QD2KC-TE, Compaq Computer Corporation, 1998.

[4] T. Austin, "DIVA: A Dynamic Approach to Microprocessor Verification," in Journal of Instruction-Level Parallelism, Vol. 2, 2000.

[5] Clark W. Barrett, David L. Dill, Jeremy R. Levitt, "Validity Checking for Combinations of Theories with Equalities," in FMCAD, 1996.

[6] R. Bryant, S. German, and M. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic," TR CMU-CS-99-115, 1999.

[7] J. Burch, and D. Dill, "Automatic Verification of Pipelined Microprocessor Control," in CAV'94.

[8] S. Chatterjee, C. Weaver, and T. Austin, "Efficient Checker Processor Design," in *Micro-33*, 2000.

[9] D. Cyrluk, "Microprocessor Verification in PVS: A Methodology and Simple Example," Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, 1993.

[10] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD based procedures for a theory of equality with uninterpreted functions," in CAV'98, June, 1998, pp. 244-255.

[11] W. A. Hunt, "FM8501: A Verified Microprocessor," Tech. Rept. 47, Institute for Computing Science, University of Texas at Austin, 1986.

[12] J. Joyce, "Formal Verification and Implementation of a Microprocessor," VLSI Specification, Verification, and Synthesis, G. Birtwistle and P.A. Subrahmanyam, eds., Kulwer Academic Publishers, 1988.

[13] K. Shimizu, D. Dill, and A. Hu. "Monitor-Based Formal Specification of PCI", in Formal Methods in Computer Aided Design, 2000.

[14] J. Silva and K. Sakallah,"GRASP-A New Search Algorithm for Satisfiability," in ICCAD, 1996.

[15] M. Srivas and M. Bickford, "Formal Verification of a pipelined microprocessor," in IEEE Software, 7(5):52-64, Sept. 1990.